CHANGES IN BASIC RELEASE 2.0

The DOS level 2.0 diskette also contains release 2.0 of Disk and Advanced BASIC. There are several differences between BASIC release 1.10 and BASIC release 2.0. This package explains those differences.

Look at the edition notice on page ii of your IBM Personal Computer *BASIC* book. If it says "Second Edition – (Revised January, 1983)" then the following pages should be added to or replaced in your BASIC book.

Use the table below to add or replace your pages. The "Description" column tells you the topic of the page(s). A check in the "R" column indicates a replacement page; a check in the "A" column indicates an added page. The page numbers in the "Old page #" column are the pages you should remove from your BASIC release 1.10 book. The "New page #" column indicates the page numbers of the new pages to be added.

(Notice that some page numbers are followed by lowercase letters. This is so that your BASIC release 2.0 pages fit into your BASIC release 1.10 book.)

Description	R	A	Old page #	New page #
Preface	✓		iii-iv	iii-iv
Summary Of Changes	✓		vi-viii	vii-x
Contents	✓		ix-xvi	xi-xviii
CHAPTER 2				
Contents	✓		2-1,2-2	2-1, 2-2
Getting BASIC Started	✓		(below)	(below)
Options on the BASIC Command	✓		2-3,2-6	2-3,2-6f
Ctrl-PrtSc		✓	-----	2-18a,2-18b
Running a BASIC Program		✓	-----	2-41,2-50
CHAPTER 3				
Contents	✓		3-1,3-2	3-1,3-2
Reserved Words	✓		3-5,3-8	3-5,3-8
Formatting Your Output		✓	-----	3-20a,3-20b
Input and Output	✓		3-33,3-36	3-33,3-36f
Graphics Modes	✓		3-41,3-43	3-41,3-43
CHAPTER 4				
Contents	✓		4-1,4-2	4-1,4-2
CHAIN/CHDIR	✓		4-37,4-38	4-37,4-38
DEF USR/DELETE	✓		4-75,4-76	4-75,4-76b
DRAW/EOF	✓		4-79,4-86	4-79,4-86
FILES	✓		4-97,4-98	4-97,4-98b
KEY/KILL	✓		4-131,4-136	4-131,4-136b
LINE	✓		4-141,4-144	4-141,4-144
LOF	✓		4-157,4-158	4-157,4-158
MID$/MKDIR	✓		4-169,4-170	4-169,4-170

Description	R	A	Old page #	New page #
ON KEY(n)	✓		4-181,4-184	4-181,4-184
ON PLAY(n)		✓	-----	4-186a,4-186d
ON TIMER		✓	-----	4-188a,4-188b
OPEN	✓		4-189,4-193	4-189,4-193c
OPEN"COM...	✓		4-194,4-200	4-194,4-200
PAINT	✓		4-203,4-204	4-203,4-204f
PLAY	✓		4-209,4-212	4-209,4-212
PLAY(n)		✓	-----	4-212a
PMAP		✓	-----	4-212b,4-212d
POINT	✓		4-213,4-214	4-213,4-214
RANDOMIZE	✓		4-235,4-238	4-235,4-238
RMDIR		✓	-----	4-248a,4-248b
SCREEN	✓		4-257,4-258	4-257,4-258
TIMER		✓	-----	4-282a,4-282b
VIEW/WAIT	✓		4-289,4-290	4-289,4-290
WINDOW/WRITE	✓		4-297,4-298	4-297,4-298b
APPENDIXES				
Contents	✓		A-1,A-4	A-1,A-4
Appendix A Messages	✓		A-5,A-18	A-5,A-26
Appendix H Conversion Tables	✓		H-1,H-2	H-1,H-2
Appendix I Technical Information and Tips	✓		I-9,I-10	I-9,I-10
Appendix K Scan Codes		✓	-----	K-1,K-2
INDEX	✓		X-1,X-10	X-1,X-14

PREFACE

The IBM Personal Computer BASIC interpreter consists of three upward compatible versions: Cassette, Disk, and Advanced. This book is a reference for all three versions of BASIC release 2.0.

Throughout this book, the term *BASIC* refers to any version of BASIC – Cassette, Disk, or Advanced.

Important: The terms *diskette*, *disk*, and *fixed disk* are used throughout this book. Where *fixed disk* is used, it applies to only the IBM non-removable fixed disk drive. The terms *disk* and *diskette* refer to diskette drives, diskettes, or the fixed disk—unless otherwise noted in the text.

The IBM Personal Computer BASIC Compiler is an optional software package available from IBM. If you have the BASIC Compiler, use the IBM Personal Computer *BASIC Compiler* book in conjunction with this book.

How to Use This Book

In order to use this manual, you should have some knowledge of general programming concepts; we are not trying to teach you how to program in this book.

The book is divided into four chapters plus a number of appendixes.

Chapter 1 is a brief overview of the three versions of IBM Personal Computer BASIC.

Chapter 2 tells you what you need to know to start using BASIC on your IBM Personal Computer. It tells you how to operate your computer using BASIC.

Chapter 3 covers a variety of topics which you need to know before you actually start programming. Much of the information pertains to data representation when using BASIC. We discuss filenames here, along with many of the special input and output features available in IBM Personal Computer BASIC.

Chapter 4 is the reference section. It contains, in alphabetical order, the syntax and semantics of every command, statement, and function in BASIC.

The appendixes contain other useful information, such as lists of error messages, ASCII codes, and math functions; plus helpful information on machine language subroutines, diskette input and output, and communications. You will also find detailed information on more advanced subjects for the more experienced programmer.

We suggest you read thoroughly chapters 2 and 3 to become familiar with BASIC. Then, while you are actually programming, you can refer to chapter 4 for the information you need about the commands or statements you are using.

SUMMARY OF CHANGES

The following changes have been made in BASIC release 2.0:

- Three enhancements have been made to the BASIC command line:
  - The optional parameter *max blocksize* has been added to the /M: switch, allowing you to load programs above the BASIC workspace.
  - The ATN, COS, EXP, LOG, SIN, SQR, and TAN functions now allow you to calculate in double precision by specifying /D on the BASIC command line.
  - You can redirect standard input and standard output by specifying <*stdin* or >*stdout* on the BASIC command line.
- Pressing Ctrl-PrtSc causes text sent to your screen to also be sent to your system printer. Ctrl-PrtSc differs from Shift-PrtSc in that all text that appears on your screen prints on your printer until you press Ctrl-PrtSc again.
- The filespec syntax has been expanded to allow the specification of a path to a device or file. All commands and statements that accept filespec also accept path. The commands that allow paths are BLOAD, BSAVE, KILL, LOAD, MERGE, NAME, RUN, and SAVE. The statements that allow paths are CHAIN and OPEN.

- The DELETE command syntax has been expanded to allow line deletions from the specified line to the end of the program.
- The PE option has been added to the OPEN "COM... statement syntax to allow for parity checking.
- The PLAY statement has two new options. For use in Advanced BASIC only.
  - >*n* raises the octave and plays note *n*.
  - <*n* lowers the octave and plays note *n*.
- The DRAW statement has two new options. For use in Advanced BASIC only.
  - TA(*n*) turns angle *n* from -360 to 360 degrees.
  - P *paint,boundary* sets figure color and border color.
- The POINT function now allows the form v=POINT(*n*), which returns the value of the current x or y graphics coordinate. For use in Advanced BASIC only.
- The RANDOMIZE statement now allows double-precision expressions. Also, you can generate a new random number without a prompt by using the TIMER function in the expression.

- The LINE statement has a new option, *style*, which uses hexadecimal values to plot a pattern of points on the screen. For use in Advanced BASIC only.

- The PAINT statement has a new option, *background*, that allows you to do paint tiling. For use in Advanced BASIC only.

- The ON KEY(n), KEY(n), and KEY statements now allow trapping of six additional definable keys, 15-20. For use in Advanced BASIC only.

- The GET and PUT statements have been enhanced to allow record numbers in the range 1 to 16,777,215 to accommodate large files with short record lengths.

- In BASIC release 2.0, EOF(0) returns the end of file condition on standard input devices used with redirection of I/O.

- In BASIC release 2.0, the LOF function returns the actual number of bytes allocated to a file.

- The graphics statements CIRCLE, LINE, PAINT, POINT, PSET, PRESET, and WINDOW now use line clipping. Lines that cross the screen or viewport are "clipped" at the boundaries of the viewing area. Only the points plotted within the screen or viewport are visible; points outside the viewing area do not appear and do not wrap around.

- Chapter 2, "How to Start and Use BASIC," has been expanded to include sections on running a BASIC program, running the SAMPLES program, running the COMM program, and running a BASIC program on another diskette.
- Chapter 3, "General Information About Programming in BASIC," has been expanded to include a section on techniques for formatting your math output. Another section, "Tree-Structured Directories," has also been added to this chapter to help you use directories in BASIC release 2.0.
- Appendix A, "Messages," now includes a Quick Reference and new messages for BASIC release 2.0.
- Appendix H, "Hexadecimal Conversion Tables," now includes a binary-to-hexadecimal conversion table.

Three new functions have been added:

- The PLAY(n) function returns the number of notes currently in the Music Background (MB) buffer. For use in Advanced BASIC only.
- The PMAP function maps an expression to world or physical coordinates. For use in Advanced BASIC only.
- The TIMER function returns the number of seconds that have elapsed since midnight or System Reset.

Four new statements have been added:

- The ON PLAY statement allows continuous music to play while a program is running. For use in Advanced BASIC only.
- The ON TIMER statement transfers control to a given line number in a BASIC program when a defined period of time has elapsed. For use in Advanced BASIC only.
- The VIEW statement lets you define a viewport (or area) within the physical limits of the screen. For use in Advanced BASIC only.
- The WINDOW statement lets you redefine the coordinates of the screen. For use in Advanced BASIC only.

Three new commands have been added:

- The CHDIR command allows you to change the current directory.
- The MKDIR command creates a directory on the specified diskette.
- The RMDIR command removes a directory from the specified diskette.

One new appendix has been added:

- Appendix K, "Keyboard Diagram and Scan Codes."

NOTES

CONTENTS

CHAPTER 2. HOW TO START AND USE BASIC

Contents

Getting BASIC Started

It's easy to start BASIC on the IBM Personal Computer:

To Start Cassette BASIC

Important: You should only use Cassette BASIC if you do not have any type of disk with your system. If you have a disk system, we recommend you use Disk or Advanced BASIC.

If your computer is off

1. Remove any diskettes from drive A and close the drive door.

2. Switch on the computer.

The words"Version C" and the release number are displayed along with the number of free bytes you have available.

If your computer is on

1. Remove any diskettes from drive A and close the drive door.

2. Press and hold down the Ctrl and Alt keys and then press the Del key.

The words "Version C" and the release number are displayed along with the number of free bytes.

To Start Disk BASIC

1. Start DOS:

   a. Insert the IBM DOS diskette in drive A.
   b. Switch on the computer.

2. Enter the command **BASIC** when DOS prompts you for a command.

   The words "Version D" and the release number are displayed along with the number of free bytes.

To Start Advanced BASIC

1. Start DOS as described above.

2. Enter the command **BASICA** when DOS prompts you for a command.

   The words "Version A" and the release number are displayed along with the number of free bytes.

Returning to DOS

Sometimes you may want to return to DOS after running a BASIC program; for example, you may want to change from Disk BASIC to Advanced BASIC.

To return to DOS from BASIC:

1. When BASIC prompts you for a command, type:

   ```
   SYSTEM
   ```

   then press the Enter key.

2. When you see the DOS prompt, DOS is ready for you to give it a command.

For more information, see "SYSTEM command" in Chapter 4.

Options on the BASIC Command

You can include options on the BASIC or BASICA command when you start Disk or Advanced BASIC. These options specify the amount of storage BASIC uses to hold programs and data, and for buffer areas. You can also ask BASIC to immediately load and run a program.

These options are not required–BASIC works just fine without them. So if you're new to BASIC, you may wish to skip over this section and go on to the next section, "Modes of Operation." Then you can refer back to this section when you become more familiar with BASIC and its capabilities.

The complete format of the **BASIC** command is:

BASIC[A] [*filespec*] [<*stdin*] [>] [>*stdout*]
[/F:*files*] [/S:*bsize*] [/C:*combuffer*] [/M:[*max workspace*]
[,*max blocksize*]] [/D]

filespec This is the file specification of a program to be loaded and run immediately. It must be a character string constant, but it should *not* be enclosed in quotation marks. In BASIC 2.0, *filespec* is expanded to allow the specification of a *path*. It should conform to the rules for specifying files described under "Naming Files" in Chapter 3. A default extension of .BAS is used if none is supplied and the length of the filename is eight characters or less. If you include *filespec*, BASIC proceeds as if a RUN *filespec* command were the first thing you entered once BASIC was ready. Note that when you specify *filespec*, the BASIC heading with the copyright notices is not displayed.

<stdin A BASIC program normally receives its input from the keyboard (standard input device). Using <*stdin* allows BASIC to receive input from the file you specify. When you use <*stdin*, you must position it before any switches. Refer to "Redirection of Standard Input and Standard Output" for more information. (For use in BASIC release 2.0 only.)

>stdout	A BASIC program normally writes its output to the screen (standard output device). Using *>stdout* allows BASIC to write output to the file or device you specify. When you use *>stdout*, you must position it before any switches. Refer to "Redirection of Standard Input and Standard Output" for more information. (For use in BASIC release 2.0 only.)
	Options beginning with a slash (/) are called switches. A *switch* is a means used to specify parameters. The following options on the BASIC command line are switches.
/F:files	This sets the maximum number of files that may be open at any one time during the running of a BASIC program. Each file requires 188 bytes of memory for the file control block, plus the buffer size specified in the /S: switch. If the /F: switch is omitted, the number of files defaults to three. The maximum value for BASIC is 15. The actual number of files that may be open simultaneously depends upon the value of the FILES= parameter in the DOS configuration file, CONFIG.SYS. The default, if not specified in CONFIG.SYS, is FILES=8. BASIC uses four files by default, leaving four files for BASIC file I/O. Therefore, /F:4 is the maximum value that you can give when FILES=8 and you want to be able to have all files open at the same time.
/S:bsize	This switch sets the buffer size for use with random files. The record length parameter on the OPEN statement may not exceed this value. The default buffer size is 128 bytes. The maximum value you may enter is 32767 bytes.

/C:combuffer

This sets the size of the buffer for receiving data when using the Asynchronous Communications Adapter. This switch has no effect unless you have an Asynchronous Communications Adapter on your system. The buffer for transmitting data with communications is always allocated to 128 bytes. The maximum value you may enter for the /C: switch is 32767 bytes. If the /C: switch is omitted, 256 bytes are allocated for the receive buffer. If you have a high-speed line, we suggest you use /C:1024. If you have two Asynchronous Communications Adapters on your system, both receive buffers are set to the size specified by this switch. You may disable RS232 support by using a value of zero (/C:0), in which case no buffer space will be reserved for communications, and communications support will not be included when BASIC is loaded.

/M:max workspace

This option sets the maximum number of bytes that may be used as BASIC workspace. BASIC is able to use a maximum of 64K bytes of memory, so the highest value you may set is 64K (hex FFFF). You can use this option to reserve space for machine language subroutines or for special data storage. You may wish to refer to "Memory Map" in Appendix I for more detailed information on how BASIC uses memory. If the /M: switch is omitted, all available memory to a maximum of 64K bytes is used.

:max blocksize

If you intend to load programs above the BASIC workspace, then use the optional parameter *max blocksize* with the **/M:** switch to reserve space for the workspace and your programs. (For use in BASIC release 2.0 only.) The parameter *max blocksize* must be in paragraphs, which are byte multiples of 16. When this parameter is omitted, 4096 (&H1000) is assumed. This allocates 65536 bytes for BASIC's DATA and STACK segment (because 4096 x 16 = 65536). If you want to allocate 65536 bytes for the BASIC workspace and 512 bytes for machine language subroutines, use **/M:,4112**. This gives you 4096 paragraphs for BASIC and 16 paragraphs for your routines. Designating **/M:2048** means that BASIC will allocate and use 32768 bytes (2048 x 16 = 32768) maximum for the BASIC workspace. Designating **/M:32000,2048** means that BASIC will allocate 32768 bytes maximum (2048 x 16 = 32768), but use only the lower 32000. This leaves 768 bytes free for program space.

/D

This switch tells BASIC that you want to use double-precision transcendental functions. (For use in BASIC release 2.0 only.) When you specify **/D**, approximately 3,000 additional bytes are used by the transcendental functions. The functions that can be converted to double precision are: ATN, COS, EXP, LOG, SIN, SQR, and TAN. If **/D** is omitted, the double-precision function is disregarded and the space is freed for program use.

Note: The options *files, max workspace, max blocksize, bsize,* and *combuffer* are all numbers that may be either decimal, octal (preceded by &0), or hexadecimal (preceded by &H).

Examples of how to use the BASIC command:

BASIC PAYROLL.BAS

This starts Disk BASIC so that it uses the defaults as just described – all memory and three files. The program PAYROLL.BAS is loaded and run.

BASICA INVEN/F:6

Here we start Advanced BASIC to use all memory and six files, and load and run INVEN.BAS. Remember, .BAS is the default extension.

BASIC /M:32768

This command starts Disk BASIC so the maximum workspace size is 32768 bytes. That is, BASIC will use only 32K bytes of memory. No more than three files will be used at one time.

BASICA B:CHKWRR.TST/F:2/M:&H9000

This command sets the maximum workspace size to hex 9000. This means Advanced BASIC can use up to 36K bytes of memory. Also, file control blocks are set up for two files, and the program CHKWRR.TST on the diskette in drive B is loaded and run.

BASIC TRANS/D

This command starts Disk BASIC. Because the /D switch is used, all transcendental functions will be calculated in double precision.

Redirection of Standard Input and Standard Output

In BASIC release 2.0, you can redirect your BASIC input and output. Standard input, normally read from the keyboard, can be redirected to any file you specify on the BASIC command line. Standard output, normally written to the screen, can be redirected to any file or device you specify on the BASIC command line.

BASIC *filespec* [<*stdin*] [>] [>*stdout*]

Examples:

1. In the following example, data read by INPUT, INPUT$, INKEY$, and LINE INPUT will continue to come from the keyboard. Data written by PRINT will go into the DATA.OUT file.

```
BASIC MYPROG >DATA.OUT
```

2. In this example, data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from the DATA.IN file. Data written by PRINT will continue to go to the screen.

```
BASIC MYPROG <DATA.IN
```

3. In the next example, data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from the MYINPUT.DAT file and data written by PRINT will go into the MYOUTPUT.DAT file.

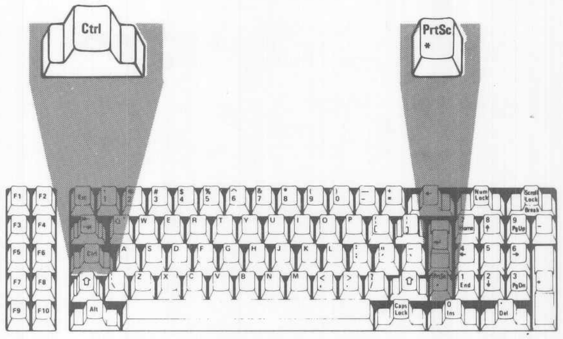
```
BASIC MYPROG <MYINPUT.DAT >MYOUTPUT.DAT
```

4. In the last example, data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from the \SALES\JOHN\TRANS file. Data output written by PRINT will be *added* to the SALES SALES.DAT file.

```
BASIC MYPROG <\SALES\JOHN\TRANS. >>\SALES\SALES.DAT
```

Notes:

- When redirected, all INPUT, INPUT$, INKEY$, and LINE INPUT statements read from the specified input file, instead of from the keyboard.
- When redirected, all PRINT statements write to the specified output file or device, instead of to the screen.
- Error messages still go to redirected standard output and to the screen. All files are closed, the program ends, and control returns to DOS.
- INPUT$, and file input from "KYBD:" still read from the keyboard.
- File output to "SCRN:" still goes to the screen.
- BASIC continues to trap keys when an ON KEY(*n*) statement is specified.
- Ctrl-PrtSc does not give a printed copy of the screen when standard output is redirected.
- Ctrl-Break goes to standard output, closes all files, and returns control to DOS.

Key(s)	Function
Ctrl-PrtSc	 In BASIC release 2.0, Ctrl-PrtSc serves as an on-off switch for any text sent to the screen to also be sent to your system printer. Press and hold the Ctrl key, press the PrtSc key, and then release both keys to print display output on the printer. Press and release both keys again to stop printing display output on the printer. Although this allows the system printer to function as a system log, it slows some operations because the computer will not continue until the printer has stopped. To get a printed copy of the screen in Cassette BASIC, see "PrtSc" under "Special Symbols" in this chapter.

NOTES

Running a BASIC Program

Two steps are involved in running a BASIC program that you have on a diskette.

The first step is getting a copy of the program transferred from the diskette into the computer's memory. This is called *loading* the program and is done with the LOAD command.

The second step is the actual performance of the program's instructions. This is called *running* the program and is done with the RUN command.

Let's go through the sequence looking at the SAMPLES program on the DOS Supplemental Programs diskette.

Running the SAMPLES Program

1. Make sure DOS is ready and A> is displayed.

2. Insert the DOS diskette into drive A if it is not already there.

3. Type:

   ```
   basic
   ```

   and press the Enter key.

4. You'll see the BASIC prompt, **Ok**.

   Now remove the DOS diskette. Insert the DOS Supplemental Programs diskette in drive A and type:

   ```
   load "samples
   ```

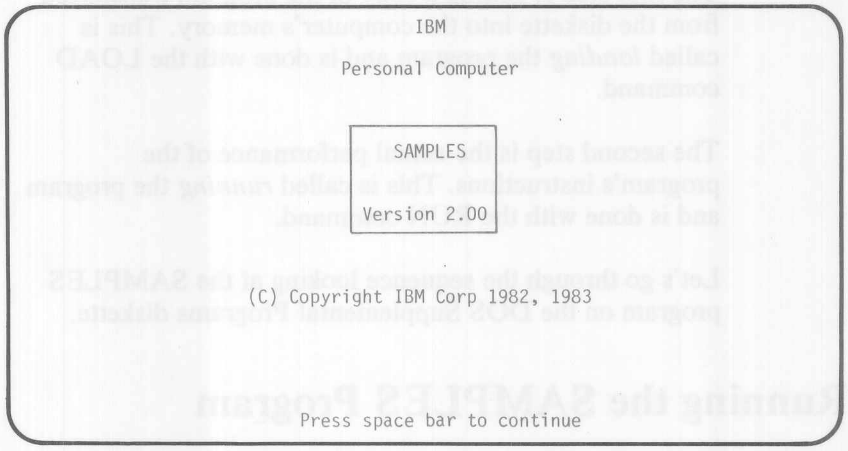
   and press the Enter key.

5. When you see **Ok**, type:

```
run
```

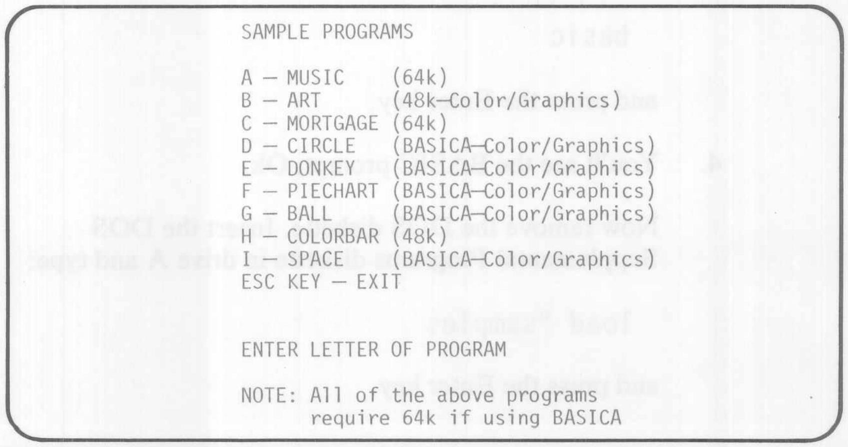
and press the Enter key.

When this screen is displayed press the space bar.



```
                    IBM
             Personal Computer

                 SAMPLES

               Version 2.00

      (C) Copyright IBM Corp 1982, 1983


        Press space bar to continue
```

6. Now you will see the *menu* screen. You select the item you want from a fixed number of choices, similar to selecting from a restaurant menu.



```
SAMPLE PROGRAMS

A — MUSIC    (64k)
B — ART      (48k—Color/Graphics)
C — MORTGAGE (64k)
D — CIRCLE   (BASICA—Color/Graphics)
E — DONKEY   (BASICA—Color/Graphics)
F — PIECHART (BASICA—Color/Graphics)
G — BALL     (BASICA—Color/Graphics)
H — COLORBAR (48k)
I — SPACE    (BASICA—Color/Graphics)
ESC KEY — EXIT


ENTER LETTER OF PROGRAM

NOTE: All of the above programs
      require 64k if using BASICA
```

Look at the remarks next to the menu items. These tell you how much computer memory you need, whether you need to use Advanced BASIC, and whether you need a Color/Graphics Monitor.

For Choices A, B, C, and H

Try choice H—COLORBAR. Type:

```
h
```

You do not need to press the Enter key. Follow the directions on the screen to see the computer display the different colors.

If you have a Color/Graphics Monitor, you can try choice B.

When you have tried any or all of these programs, press the Esc key. You will see the BASIC prompt, Ok.

For Choices D, E, F, G, and I

In order to try choices D, E, F, G, and I, you need a Color/Graphics Monitor. Let's assume that you do and are now looking at the menu.

1. Press the Esc key.

   You don't have to press the Enter key. This returns you to Disk BASIC and you will see the BASIC prompt, **Ok**.

2. Remove the DOS Supplemental Programs diskette and make sure your DOS diskette is in drive A. Then to return to DOS, type:

   ```
   system
   ```

   and press the Enter key.

   This gets you back to DOS. Now start the SAMPLES program again using Advanced BASIC instead of Disk BASIC.

3. When you see the DOS prompt, A>, type:

   ```
   basica
   ```

   and press the Enter key.

4. When you see the BASIC prompt, **Ok**, remove the DOS diskette and insert the DOS Supplemental Programs diskette in drive A. Then type:

   ```
   load "samples
   ```

   and press the Enter key.

5. When you see **Ok** again, type:

   ```
   run
   ```

   and press the Enter key.

6. The SAMPLES program menu will be displayed again.

Now you can select any choice on the menu because you are now in Advanced BASIC and should have the proper computer system.

Note: If you have the Color/Graphics Monitor Adapter, you can run the SAMPLES program from now on by starting at step 3.

Running the COMM Program

A sample telecommunications program is also provided on the DOS Supplemental Programs diskette. This program lets you establish an asynchronous communications link between your IBM Personal Computer and another IBM Personal Computer, an IBM Series/1 computer, or two communications network services.

This means that your computer can "talk" to another computer or be part of a network service. Using a network service is like being on a telephone "party line."

The COMM program will work only if you have the necessary option (Asynchronous Communications Adapter), equipment, and subscriptions. If you need help using external devices, consult your dealer.

You could also use the COMM program as a model for writing your own telecommunications program.

Let's look at the COMM program menu. (You can do this even if you don't plan to communicate with another computer.) Follow these steps:

1. Make sure Disk BASIC or Advanced BASIC is running and Ok is displayed.

2. Insert the DOS Supplemental Programs diskette in drive A if it is not already there.

3. Type:

   ```
   load "comm
   ```

   and press the Enter key.

4. Now type:

   ```
   run
   ```

   and press the Enter key.

5. The sample communications program menu is displayed.

```
                 COMMUNICATIONS MENU



           Choose one of the following:

                  1 Description of program
                  2 Dow Jones/News Retrieval
                  3 IBM Personal Computer
                  4 Series/1
                  5 THE SOURCE
                  6 Other service
                  7 End program

           Choice
```

6. You can pick options 1 or 7 now, even if you are not ready to establish communications.

   You need to have the Asynchronous Communications Adapter and the proper cabling to pick any of the other choices. Type the number of your choice and press the Enter key.

7. Each choice (except 7) will show you another menu.

   When you're through reading the information, press the F1 key. The main menu is displayed again.

8. Type:

   ```
   7
   ```

   and press the Enter key.

   You are back at BASIC again.

COMM Program Choices

Here's a short description of the COMM program choices.

1. **Description of program**

   This choice displays a screen that describes the COMM program.

2. **Dow Jones/News Retrieval**

   This choice lets you dial in to the Dow Jones/News service.

   You must have a Dow Jones/News service subscription, as well as the communications equipment, to run this choice.

3. **IBM Personal Computer**

   This choice lets your IBM Personal Computer communicate with another IBM Personal Computer.

   You must have the Asynchronous Communications Adapter and other purchased equipment to run this choice.

4. **Series/1**

   This choice lets your IBM Personal Computer communicate with an IBM Series/1 computer, running either Realtime Programming System (RPS) V5.1 or Event Driven Executive (EDX) V3.0.

5. **THE SOURCE**

   This choice lets you connect to THE SOURCE service.

   You need to get a subscription to THE SOURCE and have the adapter and purchased equipment to run this choice.

6. **Other service**

   This choice lets you describe the kind of communications your IBM Personal Computer will set up. You do this if the choices that were made in the COMM program are not correct for your case. Then you can start the communications session using the characteristics you've described.

7. **End program**

   This option ends the COMM program and takes you back to BASIC. You will then see the BASIC prompt, Ok.

Running a BASIC Program on Another Diskette

For this example, let's assume that the BASIC program you want to run is called BOWLING and it's not on your DOS diskette. Let's also assume that BOWLING is a Disk BASIC program – that is, it doesn't use the extended capabilities of Advanced BASIC.

With One Diskette Drive

1. When you see the DOS prompt, start Disk BASIC by typing:

   ```
   basic
   ```

   and pressing the Enter key.

2. The BASIC prompt appears.

   Remove your DOS diskette from drive A.

3. Insert the diskette that contains the BOWLING file into drive A.

4. Type:

   ```
   load "bowling
   ```

   and press the Enter key.

   Note: If you do not supply an extension in the command, BASIC will look for a file with the extension .BAS. In this case, BASIC will look for a file named BOWLING.BAS.

5. When you see the BASIC prompt again, type:

   ```
   run
   ```

   and press the Enter key.

6. Now the BOWLING program will perform the instructions in the program.

With Two Diskette Drives

If you have two diskette drives in your system, you could insert the diskette with the BOWLING file into drive B.

Then in step 4 above, you would type:

```
load "b:bowling
```

Notice that you have to specify the drive where the diskette with the program is located.

CHAPTER 3. GENERAL INFORMATION ABOUT PROGRAMMING IN BASIC

Contents

The following special characters have specified meanings in BASIC:

Character	*Name*
	blank
=	equal sign or assignment symbol
+	plus sign or concatenation symbol
-	minus sign
*	asterisk or multiplication symbol
/	slash or division symbol
\	backslash or integer division symbol
^	caret or exponentiation symbol
(	left parenthesis
)	right parenthesis
%	percent sign or integer type declaration character
#	number (or pound) sign, or double-precision type declaration character
$	dollar sign or string type declaration character
!	exclamation point or single-precision type declaration character
&	ampersand
,	comma
.	period or decimal point
’	single quotation mark (apostrophe), or remark delimiter
;	semicolon
:	colon or statement separator
?	question mark (PRINT abbreviation)
<	less than
>	greater than
“	double quotation mark or string delimiter
_	underline

Many characters can be printed or displayed even though they have no particular meaning to BASIC. See “Appendix G. ASCII Character Codes” for a complete list of these characters.

Reserved Words

Certain words have special meaning to BASIC. These words are called *reserved words*. Reserved words include all BASIC commands, statements, function names, and operator names. Reserved words cannot be used as variable names.

You should always separate reserved words from data or other parts of a BASIC statement by using spaces or other special characters as allowed by the syntax. That is, the reserved words must be appropriately *delimited* so that BASIC will recognize them.

The following section lists all of the reserved words in BASIC.

ABS	CVD
AND	CVI
ASC	CVS
ATN	DATA
AUTO	DATE$
BEEP	DEF
BLOAD	DEFDBL
BSAVE	DEFINT
CALL	DEFSNG
CDBL	DEFSTR
CHAIN	DELETE
CHDIR	DIM
CHR$	DRAW
CINT	EDIT
CIRCLE	ELSE
CLEAR	END
CLOSE	ENVIRON
CLS	ENVIRON$
COLOR	EOF
COM	EQV
COMMON	ERASE
CONT	ERDEV
COS	ERDEV$
CSNG	ERL
CSRLIN	ERR

ERROR
EXP
FIELD
FILES
FIX
FN*xxxxxxxx*
FOR
FRE
GET
GOSUB
GOTO
HEX$
IF
IMP
INKEY$
INP
INPUT
INPUT#
INPUT$
INSTR
INT
INTER$
IOCTL
IOCTL$
KEY
KEY$
KILL
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET
MERGE
MID$
MKDIR
MKD$
MKI$

MKS$
MOD
MOTOR
NAME
NEW
NEXT
NOT
OCT$
OFF
ON
OPEN
OPTION
OR
OUT
PAINT
PEEK
PEN
PLAY
PMAP
POINT
POKE
POS
PRESET
PRINT
PRINT#
PSET
PUT
RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RMDIR
RND
RSET
RUN
SAVE
SCREEN
SGN
SHELL
SIN
SOUND

SPACE$	TO
SPC(	TROFF
SQR	TRON
STEP	USING
STICK	USR
STOP	VAL
STR$	VARPTR
STRIG	VARPTR$
STRING$	VIEW
SWAP	WAIT
SYSTEM	WEND
TAB(	WHILE
TAN	WIDTH
THEN	WINDOW
TIME$	WRITE
TIMER	WRITE#
	XOR

Techniques for Formatting your Output

BASIC has built-in statements and functions that you can use in your programs to display numbers in the desired format and with the desired accuracy.

- Use DEFDBL to define your constants and variables as double-precision numbers to format your output. For example:

```
10 WIDTH 80
20 DEFDBL A
30 A=70#
40 PRINT A/100#,
50 A=A+1
60 IF A<100# GOTO 40
Ok
RUN
  .7         .71         .72         .73         .74
  .75        .76         .77         .78         .79
  .8         .81         .82         .83         .84
  .85        .86         .87         .88         .89
  .9         .91         .92         .93         .94
  .95        .96         .97         .98         .99
```

GENERAL INFO

- When you want your program results displayed in decimal notation, use the PRINT USING and LPRINT USING statements. These statements let you choose the format in which the results will be printed or displayed. For example, to print up to three digits to the left of the decimal point and only one to the right, you might try the following:

```
          1Ø WIDTH 8Ø
          2Ø N=1ØØ.4
          3Ø PRINT USING "###.#  ";N;
          4Ø N=N-2.5
          5Ø IF N>5 GOTO 3Ø
          Ok
          RUN

1ØØ.4  97.9  95.4  92.9  9Ø.4  87.9  85.4  82.9  8Ø.4  77.9
 75.4  72.9  7Ø.4  67.9  65.4  62.9  6Ø.4  57.9  55.4  52.9
 5Ø.4  47.9  45.4  42.9  3Ø.4  37.9  35.4  32.9  3Ø.4  27.9
 25.4  22.9  2Ø.4  17.9  15.4  12.9  1Ø.4   7.9   5.4
Ok
```

Notes:

- Avoid using both single- and double-precision numbers in the same formula because it reduces accuracy.

- Use double-precision transcendentals for greater accuracy.

Input and Output

The remainder of this chapter contains information on input and output (I/O) in BASIC. The following topics are discussed:

- Files – using BASIC files, naming files, and using devices
- Paths – using BASIC paths, naming paths, and using devices
- Tree-structured directories – using tree-structured directories
- The screen – displaying things on the screen, with an emphasis on graphics
- Other features – using clock, sound, light pen, and joy sticks

Files

A file is a collection of information which is kept somewhere other than in the random access memory of the IBM Personal Computer. For example, your information may be stored in a file on diskette or cassette. To use the information, you must *open* the file to tell BASIC where the information is. Then you may use the file for input and/or output.

BASIC supports the concept of general device I/O files. This means that any type of input/output may be treated like I/O to a file, whether you are actually using a cassette or diskette file, or are communicating with another computer.

File Number

BASIC performs I/O operations using a file number. The file number is a unique number that is associated with the actual physical file when it is opened. It identifies the path for the collection of data. A file number may be any number, variable, or expression ranging from 1 to *n*, where *n* is the maximum number of files allowed. The variable *n* is 4 in Cassette BASIC, and defaults to 3 in Disk and Advanced BASIC. It may be changed, up to a maximum of 15, by using the /F: option on the BASIC command for Disk and Advanced BASIC.

Naming Files

The physical file is described by its *file specification*, or *filespec* for short.

The file specification is a string expression of the form:

device:filename

The device name tells BASIC *where* to look for the file, and the filename tells BASIC *which* file to look for on that particular device. Sometimes you do not need both device name and filename, so specification of device and filename is optional. Note the colon (:) indicated above. Whenever you specify a device, you *must* use the colon even though a filename is not necessarily specified.

Note: File specification for communications devices is slightly different. The filename is replaced with a list of options specifying such things as line speed. Refer to "OPEN "COM... Statement" in Chapter 4 for details.

Remember that if you use a string constant for the *filespec*, you must enclose it in quotation marks. For example,

```
LOAD "B:ROTHERM.ARK"
```

In BASIC release 2.0, file specification has been enhanced to allow the specification of a *path*. In Disk and Advanced BASIC only, you can specify the path to a file.

A *path* consists of a list of directory names separated by backslashes (\). The path is a string expression of the form:

device:path

The device name tells BASIC *where* to look for the file, and the path tells BASIC *which* path to follow to get to the directory that contains a particular file. Sometimes you do not need both the device name and the path, so specification of *both* the device and the path is optional. Note the colon (:) indicated above. Whenever you specify a device, you *must* use the colon even though you may not specify a path.

You can use paths for the following commands:

```
BLOAD    KILL     OPEN
BSAVE    LOAD     RMDIR
CHAIN    MERGE    RUN
CHDIR    MKDIR    SAVE
FILES    NAME
```

Notes:

- A path may not contain more than 63 characters.
- If you place a device name anywhere other than before the path, you will see a **Bad filename** error message.
- If you use a string constant for the path, you must enclose it in quotation marks.

"B:\SALES JOHN\REPORT" – is valid

SALES\JOHN\B:REPORT – will give an error

If a filename is included, it must also be separated from the last directory name by a backslash. If a path begins with a backslash, BASIC starts its search from the root directory; otherwise, the search begins at the current directory.

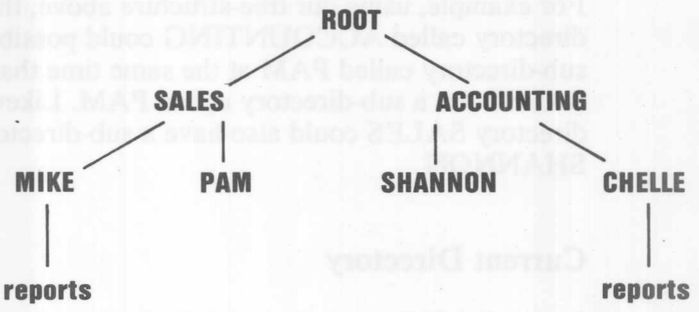
If the file is not in the current directory, you must supply BASIC with the path of directory names leading to the current directory. The path you specify can be either the path of names starting with the root directory, or the path from the current directory.

Tree-Structured Directories

Previous releases of BASIC used a simple directory structure that was adequate for managing files on diskettes. With the added support for fixed disks in BASIC release 2.0, a single disk can hold literally thousands of files.

BASIC release 2.0 gives you the ability to better organize and manage your disks by placing groups of related files in their own directories, all on the same disk or diskette.

For example, let's assume that XYZ company has two departments, sales and accounting, that share an IBM Personal Computer. All of the company's files are kept on the computer's fixed disk. The organization of the file categories could be viewed like this:



With BASIC release 2.0, you can create a directory structure that matches the organization above.

Directory Types

As in previous releases of BASIC, a single directory is created on each disk or diskette when you FORMAT it. That directory is called the *root* directory. A root directory on a diskette can hold either 64 or 112 files. The maximum number of files in a fixed disk root directory depends on the size of the DOS partition on the disk.

In addition to containing the names of files, the root directory also contains the names of other directories called *sub-directories*. Unlike the root directory, these sub-directories are actually files and can contain any number of additional files and sub-directories – limited only by the amount of available space on the diskette.

The sub-directory names are in the same format as filenames. All characters that are valid for filenames are valid for a directory name. Each directory can also contain file and directory names that also appear in other directories.

For example, using our tree-structure above, the directory called ACCOUNTING could possibly have a sub-directory called PAM at the same time that SALES has a sub-directory called PAM. Likewise, the directory SALES could also have a sub-directory called SHANNON.

Current Directory

Just as BASIC remembers a default drive, it can also remember a default directory for each drive on your system. This is called the *current* directory and is the directory that BASIC will search if you enter a filename without telling BASIC which directory the file is in. You can change the current directory by issuing the CHDIR command. (Refer to "CHDIR Command" in Chapter 4.)

For more specific information about using tree-structured directories, refer to "CHDIR Command," "MKDIR Command," and "RMDIR Command" in Chapter 4.

Device Name

The device name consists of up to four characters followed by a colon (:). The following is a complete list of device names, telling what device the name applies to, what the device can be used for (input or output), and which versions of BASIC support the device.

Device Name Chart

KYBD:	Keyboard. Input only, all versions of BASIC.
SCRN:	Screen. Output only, all versions of BASIC.
LPT1:	First printer. Output, all versions; or random, Disk and Advanced BASIC.
LPT2:	Second printer. Output or random, Disk and Advanced BASIC.
LPT3:	Third printer. Output or random, Disk and Advanced BASIC.

COMMUNICATIONS DEVICES

COM1:	First Asynchronous Communications Adapter. Input and output, Disk and Advanced BASIC.
COM2:	Second Asynchronous Communications Adapter. Input and output, Disk and Advanced BASIC.

STORAGE DEVICES

CAS1:	Cassette tape player. Input and output, all versions.
A:	First diskette drive. Input and output, Disk and Advanced BASIC.
B:	Second diskette drive. Input and output, Disk and Advanced BASIC.
C:	First fixed disk drive. Input and output, Disk and Advanced BASIC.
D:	Second fixed disk drive. Input and output, Disk and Advanced BASIC.

Note: If you have three diskette drives, then the third diskette drive is C, the first fixed disk is D, and the second fixed disk is E. Similarly, if you have four diskette drives, the fourth diskette drive is D, the first fixed disk is E, and the second fixed disk is F.

Refer to "Search Order for Adapters" in Appendix I, "Technical Information and Tips" for information on which adapters are referred to by the printer and communications device names.

Filename

The filename must conform to the following rules:

For cassette files:

- The name may not be more than eight characters long.
- The name may not contain colons, hex 00s or hex FFs (decimal 255s).

For diskette files, the name should conform to DOS conventions:

- The name may consist of two parts separated by a period (.):

  name.extension

  The *name* may be from one to eight characters long. The *extension* may be no more than three characters long.

  If *extension* is longer than three characters, the extra characters are truncated. If *name* is longer than eight characters and *extension* is not included, then BASIC inserts a period after the eighth character and uses the extra characters (up to three) for the *extension*. If *name* is longer than eight characters and an *extension* is included, then an error occurs.

- Only the following characters are allowed in *name* and *extension*:

  ```
  A through Z
  0 through 9
  (  )  {  }
  @  #  $  %  ^  &  !
  -  _  ‘  ’  /  ~  ¦
  ```

  Note: Previous releases of BASIC allowed the characters <, >, and \ to be used within a filename. However, these characters have special meaning to BASIC release 2.0 and can no longer be used in filenames.

NOTES

The statements you can use to display information in text mode are:

```
CLS            SCREEN
COLOR          WIDTH
LOCATE         WRITE
PRINT
```

The following functions and system variables may be used in text mode:

```
CSRLIN         SPC
POS            TAB
SCREEN
```

Another special feature you get in text mode if you have the Color/Graphics Monitor Adapter is multiple display pages. The Color/Graphics Monitor Adapter has a 16K-byte screen buffer, but text mode needs only 2K of that (or 4K for 80 column width). So the buffer is divided into different *pages*, which can be written on and/or displayed individually. There are 8 pages, numbered 0 to 7, in 40 column width; and 4 pages, numbered 0 to 3, in 80 column width. Refer to "SCREEN Statement" in Chapter 4 for details.

Graphics Modes

The graphics modes are available only if you have the Color/Graphics Monitor Adapter.

You can use BASIC statements to draw in two graphic resolutions:

- medium resolution: 320 by 200 points and 4 colors
- high resolution: 640 by 200 points and 2 colors

You can select which resolution you want to use with the SCREEN statement.

The statements used for graphics in BASIC are:

```
CIRCLE        PAINT
COLOR         PRESET
DRAW          PSET
GET           PUT
LINE          SCREEN
```

The only graphics function is:

```
POINT
```

Medium Resolution: There are 320 horizontal points and 200 vertical points in medium resolution. These points are numbered from left to right and from top to bottom, starting with zero. That makes the upper left corner of the screen point (0,0), and the lower right corner point (319,199). (If you are familiar with the usual mathematical method for numbering coordinates, this may seem upside-down to you.)

Medium resolution is unusual because of its color features. When you put something on the screen in medium resolution, you can specify a color number of 0, 1, 2, or 3. These colors are not fixed, as are the 16 colors in text mode. You select the actual color for color number 0 and select one of two "palettes" for the other three colors by using the COLOR statement. A palette is a set of three actual colors to be associated with the color numbers 1, 2 and 3. If you change the palette with a COLOR statement, all the colors on the screen change to match the new palette.

You can still display text characters on the screen when you are in graphics mode. The size of the characters will be the same as in text mode; that is, 25 lines of 40 characters. In medium resolution, the foreground will be color number 3, and the background will be color number 0. If you are not using a U.S. keyboard, refer to the GRAFTABL command in the DOS 2.0 book for information regarding additional character support for the Color/Graphics monitor adapter and other keyboards.

High Resolution: In high resolution there are 640 horizontal points and 200 vertical points. As in medium resolution, these points are numbered starting with zero so that the lower right corner point is (639,199).

High resolution is a little easier to describe than medium resolution since there are only two colors: black and white. Black is always 0 (zero), and white is always 1 (one).

When you display text characters in high resolution, you get 80 characters on a line. The foreground color is 1 (one) and the background color is 0 (zero). So characters will always be white on black. If you are not using a U.S. keyboard, refer to the GRAFTABL command in the DOS 2.0 book for information regarding additional character support for the Color/Graphics monitor adapter and other keyboards.

Specifying Coordinates: The graphic statements require information about where on the screen you want to draw. You give this information in the form of coordinates. Coordinates are generally in the form (*x*,*y*), where *x* is the horizontal position, and *y* is the vertical position. This form is known as *absolute form*, and refers to the actual coordinates of the point on the screen, without regard to the last point referenced.

There is another way to indicate coordinates, known as *relative form*. Using this form you tell BASIC where the point is relative to the last point referenced. This form looks like:

`STEP` (*xoffset,yoffset*)

You indicate inside the parentheses the *offset* in the horizontal and vertical directions from the last point referenced.

The "last point referenced" is set by each graphics statement. When we discuss these statements in "Chapter 4. BASIC Commands, Statements, Functions, and Variables," we will indicate what each statement sets as the last point referenced.

Note: Be careful about drawing beyond the limits of the screen with any graphics statement; it may confuse the last point referenced.

This example shows the use of both forms of coordinates:

```
100 SCREEN 1
110 PSET (200,100) 'absolute form
120 PSET STEP (10,-20) 'relative form
```

This sets two points on the screen. Their actual coordinates are (200,100) and (210,80).

Other I/O Features

Clock

You may set and read the following system variables:

DATE$	Ten-character string which is the system date, in the form *mm-dd-yyyy*.
TIME$	Eight-character string which indicates the time as *hh:mm:ss*.

Sound and Music

You can use the following statements to create sound on the IBM Personal Computer:

BEEP	Beeps the speaker.
SOUND	Makes a single sound of given frequency and duration.
PLAY	Plays music as indicated by a character string.

CHAPTER 4. BASIC COMMANDS, STATEMENTS, FUNCTIONS, AND VARIABLES

Contents

STATEMENTS

MERGE brings a section of code into the BASIC program as an overlay. That is, a MERGE operation is performed with the chaining program and the chained-to program must be an ASCII file if it is to be merged. Example:

```
CHAIN MERGE "A:OVRLAY",1ØØØ
```

After using an overlay, you will usually want to delete it so that a new overlay may be brought in. To do this, use the DELETE option, which behaves like the DELETE command. As in the DELETE command, the line numbers specified as the first and last line of the range must exist, or an **Illegal function call** error occurs. Example:

```
CHAIN MERGE "A:OVRLAY2",1ØØØ,DELETE 1ØØØ-5ØØØ
```

This example will delete lines 1000 through 5000 of the chaining program before loading in the overlay (chained-to program). The line numbers in *range* are affected by the RENUM command.

Notes:

1. The CHAIN statement leave files open.
2. The CHAIN statement with MERGE option preserves the current OPTION BASE setting.
3. Without MERGE, CHAIN does not preserve variable types or user-defined functions for use by the chained-to program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.
4. The CHAIN statement does a RESTORE before running the chained-to program. This means that the read operation does not continue where it left off in the chaining program.

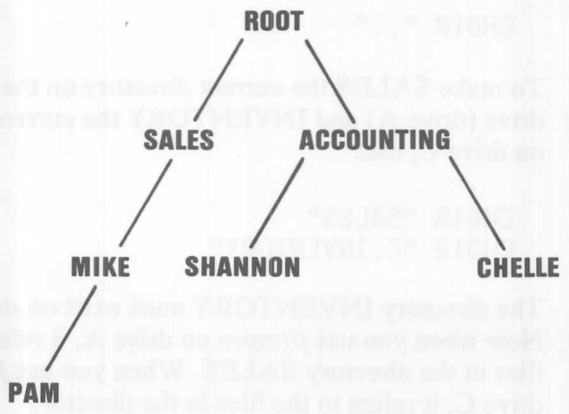
CHDIR Command

Purpose:	Allows you to change the current directory. For use in BASIC release 2.0 only.			
Versions:	Cassette	Disk	Advanced	Compiler
		***	***	

Format: CHDIR *path*

Remarks: *path* is a string expression, not exceeding 63 characters, identifying the new directory that will become the current directory. For more information on paths refer to "Naming Files" and "Tree-Structured Directories" in Chapter 3.

Examples:



The following examples refer to the tree-structure above.

To change to the root directory from any sub-directory, use:

```
CHDIR "\"
```

To change to the directory PAM from the root directory, use:

```
CHDIR "SALES\MIKE\PAM"
```

To change to the directory CHELLE from the directory ACCOUNTING, use:

```
CHDIR "CHELLE"
```

To change from the directory SALES to the directory MIKE, use:

```
CHDIR ".."
```

To make SALES the current directory on the current drive (drive A) and INVENTORY the current directory on drive C, use:

```
CHDIR "SALES"
CHDIR "C:INVENTORY"
```

The directory INVENTORY must exist on drive C. Now when you use *filespec* on drive A, it refers to the files in the directory SALES. When you use *filespec* on drive C, it refers to the files in the directory INVENTORY.

NOTES

STATEMENTS

CHR$ Function

Purpose:	Converts an ASCII code to its character equivalent.			
Versions:	Cassette	Disk	Advanced	Compiler
	***	***	***	***

Format: *v$* = CHR$(*n*)

Remarks: *n* must be in the range 0 to 255.

The CHR$ function returns the one-character string with ASCII code *n*. (ASCII codes are listed in "Appendix G. ASCII Character Codes.") CHR$ is commonly used to send a special character to the screen or printer. For instance, the BEL character, which beeps the speaker, might be included as CHR$(7) as a preface to an error message (instead of using BEEP). Look under "ASC Function," earlier in this chapter, to see how to convert a character back to its ASCII code.

Example:

```
Ok
PRINT CHR$(66)
B
Ok
```

The next example sets function key F1 to the string "AUTO" joined with Enter. This is a good way to set the function keys so the Enter is automatically done for you when you press the function key.

```
Ok
Key 1,"AUTO"+CHR$(13)
Ok
```

DEF USR Statement

Purpose: Specifies the starting address of a machine language subroutine, which is later called by the USR function.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: DEF USR[*n*]=*offset*

Remarks:

n may be any digit from 0 to 9. It identifies the number of the USR routine whose address is being specified. If *n* is omitted, DEF USR0 is assumed.

offset is an integer expression in the range 0 to 65535. The value of *offset* is added to the current segment value to obtain the actual starting address of the USR routine. See "DEF SEG Statement" in this chapter.

It is possible to redefine the address for a USR routine. Any number of DEF USR statements may appear in a program, thus allowing access to as many subroutines as necessary. The most recently executed value is used for the offset.

Refer to "Appendix C. Machine Language Subroutines" for complete information.

Example:

```
200 DEF SEG = 0
210 DEF USR0=24000
500 X=USR0(Y+2)
```

This example calls a routine at absolute location 24000 in memory.

DELETE Command

Purpose: Deletes program lines.

Versions:	Cassette	Disk	Advanced	Compiler
	***	***	***	

Format: DELETE [*line1*] [−*line2*]

DELETE [*line1*−]

Remarks:

line1	is the line number of the first line to be deleted.
line2	is the line number of the last line to be deleted.

The DELETE command erases the specified range of lines from the program. BASIC always returns to command level after a DELETE is executed.

DELETE *line1*− deletes all lines from the specified line number through the end of the program. This form is for use in BASIC release 2.0 only.

A period (.) may be used in place of the line number to indicate the current line. If you specify a line number that does not exist in the program, an **Illegal function call** error occurs.

Examples: This example deletes line 40:

```
DELETE 40
```

This example deletes lines 40 through 100, inclusive:

```
DELETE 40-100
```

This example deletes line 40 through the end of the program:

```
DELETE 40-
```

The last example deletes all lines up to and including line 40:

```
DELETE -40
```

NOTES

Purpose:	Draws an object as specified by *string*.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Graphics mode only.

Format: DRAW *string*

Remarks: You use the DRAW statement to draw using a *graphics definition language*. The language commands are contained in the string expression *string*. The string defines an object, which is drawn when BASIC executes the DRAW statement. During execution, BASIC examines the value of *string* and interprets single letter commands from the contents of the string. These commands are detailed below:

The following movement commands begin movement from the last point referenced. After each command, the last point referenced is the last point the command draws.

U n	Move up.
D n	Move down.
L n	Move left.
R n	Move right.
E n	Move diagonally up and right.

F n	Move diagonally down and right.
G n	Move diagonally down and left.
H n	Move diagonally up and left.

n in each of the preceding commands indicates the distance to move. The number of points moved is *n* times the scaling factor (set by the **S** command).

M x,y	Move absolute or relative. If *x* has a plus sign (+) or a minus sign (−) in front of it, it is relative. Otherwise, it is absolute.

The aspect ratio of your screen determines the spacing of the horizontal, vertical, and diagonal points. For example, the standard aspect ratio of 4/3 indicates that the horizontal axis of the screen is 4/3 as long as the vertical axis. You can use this information to determine how many vertical points are equal in length to how many horizontal points.

For example, in medium resolution there are 320 horizontal points and 200 vertical points. That means 8 horizontal points are equal in length to 5 vertical points if the screen aspect ratio is 1/1. If the aspect ratio is different, you multiply the number of vertical points by the aspect ratio. For example, using the standard aspect ratio of 4/3, in medium resolution 8 horizontal points are equal in length to 20/3 vertical points, or 24 horizontal equal 20 vertical. That is:

```
DRAW "U8Ø R96 D8Ø L96"
```

produces a square in medium resolution. Following similar reasoning, again with the standard screen aspect ratio of 4/3, in high resolution 48 horizontal points are equal in length to 20 vertical points.

The following two prefix commands may precede any of the above movement commands.

B	Move, but don't plot any points.
N	Move, but return to the original position when finished.

The following commands are also available:

A n	Set angle *n*. *n* may range from 0 to 3, where 0 is 0 degrees, 1 is 90, 2 is 180, and 3 is 270. Figures rotated 90 or 270 degrees are scaled so that they appear the same size as with 0 or 180 degrees on a display screen with standard aspect ratio 4/3.

TA n	Turn angle *n*. *n* can range from −360 to +360. If *n* is positive (+), the angle turns counterclockwise. If *n* is negative (−), the angle turns clockwise. Values entered that are outside of the range −360 to +360 cause an **Illegal function call** error. This command is for use in BASIC release 2.0 only.
C n	Set color *n*. *n* may range from 0 to 3 in medium resolution, and 0 to 1 in high resolution. In medium resolution, *n* selects the color from the current palette as defined by the COLOR statement. 0 is the background color. The default is the foreground color, color number 3. In high resolution, *n* equal to 0 (zero) indicates black, and the default of 1 (one) indicates white.
S n	Set scale factor. *n* may range from 1 to 255. *n* divided by 4 is the scale factor. For example, if *n*=1, then the scale factor is 1/4. The scale factor multiplied by the distances given with the U, D, L, R, E, F, G, H, and relative M commands gives the actual distance moved. The default value is 4, so the scale factor is 1.
X variable;	Execute substring. This allows you to execute a second string from within a string.

P paint,boundary
Set figure color to *paint* and border color to *boundary*. The *paint* parameter can range from 0 to 3. In medium resolution, this color is the color from the current palette as defined by the COLOR statement. In high resolution, 0 indicates black and 1 indicates white. The *boundary* parameter is the border color of the figure to be filled in, in the range 0 to 3 as described in *boundary*. You must specify both *paint* and *boundary* or an error will occur. this command is for use in BASIC release 2.0 only and does not support tile painting.

In all these commands, the *n, x,* or *y* argument can be a constant like 123 or it can be =*variable;* where *variable* is the name of a numeric variable. The semicolon (;) is required when you use a variable this way, or in the **X** command. Otherwise, a semicolon is optional between commands. Spaces are ignored in *string*. For example, you could use variables in a move command this way: M+X1; −=X2;

You can also specify variables in the form VARPTR$(*variable*), instead of =*variable;*. This is the only form that can be used in compiled programs. For example:

One Method **Alternative Method**

```
DRAW "XA$;"      DRAW "X"+VARPTR$(A$)
DRAW "S=SCALE;" DRAW "S="+VARPTR$(SCALE)
```

The **X** command can be a very useful part of DRAW, because you can define a part of an object separate from the entire object. For example, a leg could be part of a man. You can also use **X** to draw a string of commands more than 255 characters long.

When coordinates which are out of range are given to DRAW, the coordinate which is out of range is given the closest valid value. In other words, the negative values become zero and Y values greater than 199 become 199. X values greater than 639 become 639. X values greater than 319 in medium resolution wrap to the next horizontal line. An exception to this is the **TA** command. If the values you enter with **TA** are out of range, an **Illegal function call** error occurs.

Examples: To draw a box:

```
5 SCREEN 1
1Ø A=2Ø
2Ø DRAW "U=A,R=A,D=A,L=A;"
```

To draw a box and paint the interior:

```
1Ø DRAW "U5ØR5ØD5ØL5Ø" 'Draw a box
2Ø DRAW "BE1Ø" 'Move up and right into box
3Ø DRAW "P1,3" 'Paint interior
```

To draw a triangle:

```
10 SCREEN 1
20 DRAW "E15 F15 L30"
```

To create a "shooting star":

```
10 SCREEN 1,0: COLOR 0,0: CLS
20 DRAW "BM300,25" ' initial point
30 STAR$+"M+7,17 M-17,-12 M+20,0 M-17,12 M+7,-17"
40 FOR SCALE=1 TO 40 STEP 2
50 DRAW"C1;S=SCALE; BM-2,0;XSTAR$;"
60 NEXT
```

To draw some "spokes":

```
10 FOR D=0 to 360 STEP 10
20 DRAW "TA=D; NU50"
30 NEXT D
```

EDIT Command

Purpose: Displays a line for editing.

Versions:	Cassette	Disk	Advanced	Compiler
	***	***	***	

Format: EDIT *line*

Remarks: *line* is the line number of a line existing in the program. If there is no such line, an **Undefined line number** error occurs. A period (.) can be used for the line number to refer to the current line.

The EDIT statement simply displays the line specified and positions the cursor under the first digit of the line number. The line may then be modified as described under "The BASIC Program Editor" in Chapter 2.

A period (.) can be used for the line number to refer to the current line. For example, if you have just entered a line and wish to go back and change it, the command **EDIT.** will redisplay the line for editing.

LIST may also be used to display program lines for changing. Refer to "LIST Command" in this chapter.

END Statement

Purpose: Terminates program execution, closes all files, and returns to command level.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: END

Remarks: END statements may be placed anywhere in the program to terminate execution. END is different from STOP in two ways:

- END does not cause a Break message to be printed.
- END closes all files.

An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

Example:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

This example ends the program if K is greater than 1000; otherwise, the program branches to line number 20.

EOF Function

Purpose:	Indicates an end of file condition.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Format: v = EOF(*filenum*)

Remarks: *filenum* is the number specified on the OPEN statement.

The EOF function is useful for avoiding an **Input past end** error. EOF returns −1 (true) if end of file has been reached on the specified file. A 0 (zero) is returned if end of file has not been reached.

EOF is significant only for a file opened for sequential input from diskette or cassette, or for a communications file. A −1 for a communications file means that the buffer is empty.

Example:

```
10 OPEN "DATA" FOR INPUT AS #1
20 C=0
30 IF EOF(1) THEN END
40 INPUT #1,M(C)
50 C=C+1: GOTO 30
```

This example reads information from the sequential file named "DATA". Values are read into the array M until end of file is reached.

In BASIC 2.0, EOF(0) returns the end of file condition on standard input devices used with redirection of I/O.

FILES Command

Purpose:	Displays the names of files residing on the current directory of a diskette. The FILES command in BASIC is similar to the DIR command in DOS.			
Versions:	Cassette	Disk ***	Advanced ***	Compiler (**)

Format: FILES [*filespec*]

Remarks: *filespec* is a string expression for the file specification as explained under "Naming Files" in Chapter 3. If *filespec* is omitted, all the files on the current directory of the DOS default drive will be listed.

All files matching the filename are displayed. The filename may contain question marks (?). A question mark matches any character in the name or extension. An asterisk (*) as the first character of the name or extension will match any name or any extension.

If a drive is specified as part of *filespec*, files which match the specified filename on the current directory of that drive are listed. Otherwise, the DOS default drive is used.

Example:
```
FILES
```

This displays all files on the current directory of the DOS default drive.

```
FILES "*.BAS"
```

This displays all files with an extension of .BAS on the current directory of the DOS default drive.

```
FILES "B:*.*"
```

This displays all files on drive B.

```
FILES "TEST??.BAS"
```

This lists each file on the current directory of the DOS default drive that has a filename beginning with TEST followed by up to two other characters, and an extension of .BAS.

In BASIC release 2.0, another way to list all of the files on the current directory of drive B is:

```
FILES "B:"
```

In addition to listing all the files on the current directory of the drive, BASIC also displays the current directory name and the number of bytes free.

When using tree-structured directories in BASIC release 2.0, each sub-directory contains two special entries – you will see them listed when you use the FILES command to list a sub-directory. The first contains a single period instead of a filename. It identifies this "file" as a sub-directory. The second entry contains two periods instead of a filename, and is used to locate the higher level directory that defines this sub-directory (the "parent" of the sub-directory).

```
FILES
A:\LEVEL1

        .   <DIR>    ..  <DIR>

  32824 Bytes free
```

This example lists all files in the current sub-directory called LEVEL1 on drive A. Note that the directory is empty.

```
FILES "LVL1\"
```

The FILES command can also be used to list files in other directories. The example above lists all files in the sub-directory LVL1. The backslash must be used after the directory name.

```
FILES "LVL2\*.BAS"
```

This example lists all files in the directory LVL2 with an extension of .BAS.

NOTES

KEY Statement

Purpose: Sets or displays the soft keys.

Versions:	Cassette	Disk	Advanced	Compiler
	***	***	***	(**)

Format:

KEY ON

KEY OFF

KEY LIST

KEY *n*, *x$*

Remarks:

n is the function key number in the range 1 to 10. In BASIC release 2.0, *n* is a function key number in the range 1 to 20.

x$ is a string expression which will be assigned to the key. (Remember to enclose string *constants* in quotation marks.)

The KEY statement allows function keys to be designated *soft keys*. That is, you can set each function key to automatically type any sequence of characters. A string of up to 15 characters may be assigned to any one or all the ten function keys. When the key is pressed, the string will be input to BASIC.

Initially, the soft keys are assigned the following values:

F1	LIST	F2	RUN ←
F3	LOAD“	F4	SAVE“
F5	CONT ←	F6	,“LPT1:”←
F7	TRON ←	F8	TROFF ←
F9	KEY	F10	SCREEN 0,0,0 ←

The arrow (←) indicates Enter.

KEY ON causes the soft key values to be displayed on the 25th line. When the width is 40, five of the ten soft keys are displayed. When the width is 80, all ten are displayed. In either width, only the first six characters of each value are displayed. ON is the default state for the soft key display.

KEY OFF erases the soft key display from the 25th line, making that line available for program use. It does not disable the function keys.

After turning off the soft key display with KEY OFF, you can use LOCATE 25,1 followed by PRINT to display anything you want on the bottom line of the screen. Information on line 25 is not scrolled, as are lines 1 through 24.

KEY LIST lists all ten soft key values on the screen. All 15 characters of each value are displayed.

KEY *n, x$* assigns the value of *x$* to the function key specified (1 to 10). *x$* may be 1 to 15 characters in length. If it is longer than 15 characters, only the first 15 characters are assigned.

Assigning a null string (string of length zero) to a soft key disables the function key as a soft key.

If the value entered for *n* is not in the range 1 to 10, an Illegal function call error occurs. The previous key string assignment is retained.

When a soft key is pressed, the INKEY$ function returns one character of the soft key string each time it is called. If the soft key is disabled, INKEY$ returns a two character string. The first character is binary zero, the second is the key scan code, as listed in "Appendix G. ASCII Character Codes."

Note: To avoid complications on the input buffer in Cassette BASIC, you should execute:

```
DEF SEG: POKE 1Ø6,Ø
```

after reassigning any soft keys and after INKEY$ has received the last character you want from a soft key string. This POKE is not required in Disk or Advanced BASIC.

KEY Statement

In Advanced BASIC 2.0, there are six additional definable key traps. This lets you trap any Ctrl, Shift, or super-shift key. These additional keys are defined by the statement:

KEY *n*,CHR$(*shift*)+CHR$(*scan code*)

n is a numeric expression in the range 15 to 20.

shift is a numeric value that corresponds to the hex value for the latched keys. The hex values for each key are:

Caps Lock	&H40
Num Lock	&H20
Alt	&H08
Ctrl	&H04
Shift	&H01, &H02, &H03

Note that key trapping assumes that the left and right Shift keys are the same, so you can use a value of &H01, &H02, or &H03 (the sum of hex 01 and hex 02) to denote a Shift key.

You can also add multiple shift states together, such as the Ctrl and Alt keys added together. Shift state values *must* be in hex.

scan code is a number in the range 1 to 83 that identifies the key to be trapped. See Appendix K, "Keyboard Diagram and Scan Codes," for a complete table of scan codes and their associated key positions.

When you trap keys, they are processed in the following order:

1. Ctrl-PrtSc, which activates the line printer, is processed first. Even if Ctrl-PrtSc is defined as a trappable key, it can still be pressed to get a printed copy of the screen.

2. Next, the function keys F1 to F10, Cursor Up, Cursor Down, Cursor Right, and Cursor Left (1-14) are processed. Setting scan codes 59 to 68, 72, 75, 77, or 80 as key traps has no effect, because they are considered to be predefined.

3. Last, the keys you define for 15 to 20 are processed.

Notes:

- Trapped keys do not go into the keyboard buffer.
- Be careful when you trap Ctrl-Break and Ctrl-Alt-Del, because unless you have a test in your trap routine, you will have to turn the power off to stop your program.

See the following section, "KEY(n) Statement," to see how to enable and disable function key trapping in Advanced BASIC.

Examples:
```
50 KEY ON
```

displays the soft keys on the 25th line.

```
200 KEY OFF
```

erases soft key display. The soft keys are still active, but not displayed.

```
10 KEY 1,"FILES"+CHR$(13)
```

assigns the string “FILES”+Enter to soft key 1. This is a way to assign a commonly used command to a function key.

```
20 KEY 1,""
```

disables function key 1 as a soft key.

```
100 KEY 15, CHR$(&H40)+CHR$(25)
100 ON KEY(15) GOSUB 1000
120 KEY(15) ON
```

sets up a Key trap for capital P. Note that all three KEY statements – KEY, KEY(n), and ON KEY – are used with key trapping.

```
200 KEY 20, CHR$(&H04+&H03)+CHR$(30)
210 ON KEY(20) GOSUB 2000
220 KEY(20) ON
```

sets up a Key trap for Ctrl-Shift A. Notice that the hex values for Ctrl (&H04) and Shift (&H03) are added together to get the shift state.

NOTES

KEY(n) Statement

Purpose: Activates and deactivates trapping of the specified key in a BASIC program. See "ON KEY(n) Statement" in this chapter.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: KEY(*n*) ON

KEY(*n*) OFF

KEY(*n*) STOP

Remarks: *n* is a numeric expression in the range 1 to 20, and indicates the key to be trapped:

1-10	function keys F1 to F10
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down
15-20	keys defined by the form: KEY *n*,CHR$(*shift*)+CHR$(*scan code*). Keys 15-20 can be trapped only in BASIC release 2.0.

KEY(*n*) ON must be executed to activate trapping of function key or cursor control key activity. After KEY(*n*) ON, if a non-zero line number was specified in the ON KEY(*n*) statement then every time BASIC starts a new statement it will check to see if the specified key was pressed. If so it will perform a GOSUB to the line number specified in the ON KEY(*n*) statement. A KEY(*n*) statement cannot precede an ON KEY(*n*) statement.

If KEY(*n*) is OFF, no trapping takes place and even if the key is pressed, the event is not remembered.

Once a KEY(*n*) STOP statement has been executed, no trapping will take place. However, if you press the specified key your action is remembered so that an immediate trap takes place when KEY(*n*) ON is executed.

KEY (*n*) ON has no effect on whether the soft key values are displayed at the bottom of the screen.

If you use a KEY(*n*) statement in Cassette or Disk BASIC, you will get a Syntax error. Refer to the previous section, "KEY Statement," for an explanation of the KEY statement without the (*n*).

KILL Command

Purpose: Deletes a file from a diskette. The KILL command in BASIC is similar to the ERASE command in DOS.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

Format: KILL *filespec*

Remarks: *filespec* is a valid file specification as explained under "Naming Files" in Chapter 3. The device name must be a diskette drive. If the device name is omitted, the DOS default drive is used.

In BASIC release 2.0, filespec can also be a path, as explained under "Naming Files" in Chapter 3.

KILL can be used for all types of diskette files. The name must include the extension, if one exists. For example, you may save a BASIC program using the command

```
SAVE "TEST"
```

BASIC supplies the extension **.BAS** for the SAVE command, but not for the KILL command. If you want to delete that program file later, you must say KILL "TEST.BAS", not KILL"TEST".

If a KILL statement is given for a file that is currently open, a **File already open** error occurs.

Example: To delete the file named "DATA1" on drive A, you might use:

```
200 KILL "A:DATA1"
```

To delete the file "PROG.BAS" in the LEVEL2 sub-directory, you might use:

```
KILL "LEVEL1\LEVEL2\PROG.BAS"
```

Note that KILL can only be used to delete files. The RMDIR command must be used to remove directories.

NOTES

Purpose:	Draws a line or a box on the screen.			
Versions:	Cassette	Disk	Advanced	Compiler
	***	***	***	(**)

Graphics mode only.

Format: LINE [(*x1,y1*)] −(*x2,y2*) [,[*color*] [,B[F]] [,*style*]]

Remarks: *(x1,y1), (x2,y2)*

are coordinates in either absolute or relative form. (See "Specifying Coordinates" under "Graphics Modes" in Chapter 3.)

color is the color number in the range 0 to 3. In medium resolution, *color* selects the color from the current palette as defined by the COLOR statement. 0 is the background color. The default is the foreground color, a *color* of 0(zero) indicates black, and the default of 1 (one) indicates white.

style is a 16-bit integer mask used to put points on the screen. The *style* option is used for normal lines and boxes, but cannot be used with filled boxes (BF). Using *style* with BF results in a Syntax error. This technique, called line styling, is for use in BASIC release 2.0 only.

The simplest form of LINE is:

```
LINE -(X2,Y2)
```

This will draw a line from the last point referenced to the point (X2,Y2) in the foreground color.

We can include a starting point also:

```
LINE (Ø,Ø)-(319,199) 'diagonal down screen
LINE (Ø,1ØØ)-(319,1ØØ) 'bar across screen
```

We can indicate the color to draw the line in:

```
LINE (1Ø,1Ø)-(2Ø,2Ø),2 'draw in color 2

1 'draw random lines in random colors
1Ø SCREEN 1,Ø,Ø,Ø: CLS
2Ø LINE -(RND*319,RND*199),RND*4
3Ø GOTO 2Ø

1 'alternating pattern - line on, line off
1Ø SCREEN 1,Ø,Ø,Ø: CLS
2Ø FOR X=Ø TO 319
3Ø LINE (X,Ø)-(X,199),X AND 1
4Ø NEXT
```

The next argument to LINE is **B** (box), or **BF** (filled box). We can leave out *color* and include the argument:

```
LINE (Ø,Ø)-(1ØØ,1ØØ),,B 'box in foreground
```

or we may include the color:

```
LINE (Ø,Ø)-(1ØØ,1ØØ),2,BF 'filled box color 2
```

The **B** tells BASIC to draw a rectangle with the points (*x1,y1*) and (*x2,y2*) as opposite corners. This avoids having to give the four LINE commands:

```
LINE (X1,Y1)-(X2,Y1)
LINE (X1,Y1)-(X1,Y2)
LINE (X2,Y1)-(X2,Y2)
LINE (X1,Y2)-(X2,Y2)
```

which perform the equivalent function.

The **BF** means draw the same rectangle as **B**, but also fill in the interior points with the selected color.

The last argument to line is *style*. LINE uses the current bit in *style* to plot (or store) points on the screen. If the bit is 0 (zero), no point is plotted. If the bit is 1 (one), a point is plotted. After each point, the next bit position in *style* is selected. When the last bit position in *style* is selected, LINE "wraps around" and begins with the first bit position again.

Note that a 0 (zero) bit skips over the point on the screen, but it does not erase the existing point. You may want to draw a background line before a styled line to force a known background.

The *style* option can be used to draw a dotted line across the screen by plotting (storing) every other point. Because *style* is 16 bits wide, the pattern for a dotted line looks like this:

1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0

This is equal to AAAA in hexadecimal notation. For help in choosing the correct hexadecimal values, see Appendix H, "Hexadecimal Conversion Tables."

LINE Statement

Examples: To draw a dotted line:

```
10 SCREEN 1,0
20 LINE (0,0)-(319,199),,,&HAAAA
```

To draw a cyan box with dashes:

```
10 SCREEN 1,0
20 LINE (0,0)-(100,100),1,B,&HCCCC
```

In BASIC release 1.1, out-of-range coordinates given to the LINE statement are given the closest valid value. In other words, the negative values become zero and Y values greater than 199 become 199. X values greater than 639 become 639. X values greater than 319 in medium resolution wrap to the next horizontal line. This is called *wraparound* because the image goes off the viewing surface and "wraps around" to reappear.

In BASIC release 2.0, out-of-range coordinates are not visible on the viewing surface. This is called *line clipping* because the image that is outside of the coordinate range is "clipped" at the boundaries of the viewing surface.

The last point referenced after a LINE statement is point (*x2,y2*). If you use the relative form for the second coordinate, it is relative to the first coordinate. For example,

```
LINE (1ØØ,1ØØ)-STEP (1Ø,-2Ø)
```

will draw a line from (100,100) to (110,80).

This example will draw random filled boxes in random colors.

```
1Ø CLS
2Ø SCREEN 1,Ø: COLOR Ø,Ø
3Ø LINE -(RND*319,RND*199),RND*2+1,BF
4Ø GOTO 3Ø 'boxes will overlap
```

LINE INPUT Statement

Purpose:	Reads an entire line (up to 254 characters) from the keyboard into a string variable, ignoring delimiters.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LINE INPUT[;] [*"prompt"*;] *stringvar*

Remarks:

"prompt" is a string constant that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of the prompt string.

stringvar is the name of the string variable or array element to which the line will be assigned. All input from the end of the prompt to the Enter is assigned to *stringvar*. Trailing blanks are ignored.

In Disk and Advanced BASIC, if LINE INPUT is immediately followed by a semicolon, then pressing Enter to end the input line does not produce a carriage return/line feed sequence on the screen. That is, the cursor remains on the same line as your response.

You can exit LINE INPUT by pressing Ctrl-Break. BASIC returns to command level and displays **Ok**. You may then enter CONT to resume execution at the LINE INPUT.

Example: See example in the next section, "LINE INPUT # Statement."

Example:

```
10 LOCATE 1,1
```

Moves the cursor to the home position in the upper left-hand corner of the screen.

```
20 LOCATE ,,1
```

Makes the blinking cursor visible; its position remains unchanged.

```
30 LOCATE ,,,7
```

Position and cursor visibility remain unchanged. Sets the cursor to display at the bottom of the character on the Color/Graphics Monitor Adapter (starting and ending on scan line 7).

```
40 LOCATE 5,1,1,0,7
```

Moves the cursor to line 5, column 1. Makes the cursor visible, covering the entire character cell on the Color/Graphics Monitor Adapter, starting at scan line 0 and ending on scan line 7.

LOF Function

Purpose: Returns the number of bytes allocated to the file (length of the file).

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: *v* = LOF(*filenum*)

Format: *filenum* is the file number used when the file was opened.

For diskette files created by BASIC 1.10, LOF will return a multiple of 128. For example, if the actual data in the file is 257 bytes, the number 384 will be returned. For diskette files created outside BASIC (for example, by using EDLIN) and for files created by BASIC 2.0, LOF returns the actual number of bytes allocated to the file.

For communications, LOF returns the amount of free space in the input buffer. That is, *size*-LOC(*filenum*), where *size* is the size of the communications buffer, which defaults to 256 but may be changed with the /C: option on the BASIC command. Use of LOF may be used to detect when the input buffer is getting full. In practicality, LOC is adequate for this purpose, as demonstrated in the example in "Appendix F. Communications."

Example: These statements will get the last record of the file named BIG, assuming BIG was created with a record length of 128 bytes:

```
10 OPEN "BIG" AS #1
20 GET #1,LOF(1)/128
```

Example: The first example uses the MID$ function to select the middle portion of the string B$.

```
Ok
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
RUN
GOOD EVENING
Ok
```

The next example uses the MID$ statement to replace characters in the string A$.

```
Ok
10 A$="MARATHON, GREECE"
20 MID$(A$,11)="FLORIDA"
30 PRINT A$
RUN
MARATHON, FLORIDA
Ok
```

Note in the second example how the length of A$ was not changed.

MKDIR Command

Purpose: Creates a directory on the specified diskette. For use in BASIC release 2.0 only.

Versions:	Cassette	Disk	Advanced	Compiler
		***	***	

Format: MKDIR *path*

Remarks:

path	is a string expression, not exceeding 63 characters, that identifies the new directory to be created. For more information about paths refer to "Naming Files" and "Tree-Structured Directories" in Chapter 3.

Examples: From the root directory, create a sub-directory called SALES.

```
MKDIR "SALES"
```

From the root directory, create a sub-directory called MIKE under the directory SALES.

```
MKDIR "SALES\MIKE"
```

From the root directory, create a sub-directory called PAM under the directory MIKE.

```
MKDIR "SALES\MIKE\PAM"
```

From the root directory, create a sub-directory called ACCOUNTING.

```
MKDIR "ACCOUNTING"
```

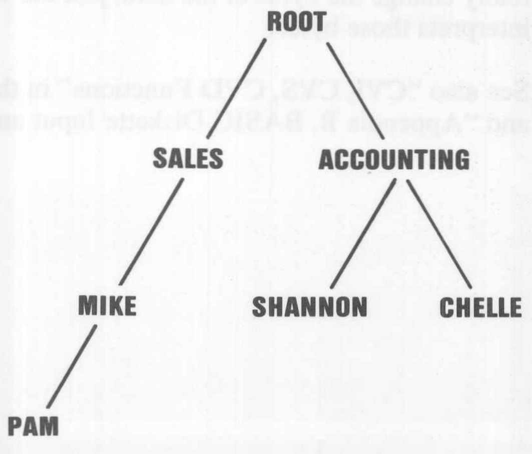
Make ACCOUNTING the current directory, then create two sub-directories called SHANNON and CHELLE.

```
CHDIR "ACCOUNTING"
MKDIR "SHANNON"
MKDIR "CHELLE"
```

The same structure could have been created from the root by entering:

```
MKDIR "ACCOUNTING\SHANNON"
MKDIR "ACCOUNTING\CHELLE"
```

By following the above examples, you have created a tree structure that looks like this:



MKI$, MKS$, MKD$ Functions

Purpose: Convert numeric type values to string type values.

Versions:	Cassette	Disk	Advanced	Compiler
		***	***	***

Format: *v$* = MKI$(*integer expression*)

*v$* = MKS$(*single-precision expression*)

*v$* = MKD$(*double-precision expression*)

Remarks: Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single-precision number to a 4-byte string. MKD$ converts a double-precision number to an 8-byte string.

These functions differ from STR$ because they do not really change the bytes of the data, just the way BASIC interprets those bytes.

See also "CVI, CVS, CVD Functions" in this chapter and "Appendix B. BASIC Diskette Input and Output."

Example: The first example branches to line 150 if L−1 equals 1, to line 300 if L−1 equals 2, to line 320 if L−1 equals 3, and to line 390 if L−1 equals 4. If L−1 is equal to 0 (zero) or is greater than 4, then the program just goes on to the next statement.

```
100 ON L-1 GOTO 150,300,320,390
```

The next example shows how to use an ON-GOSUB statement.

```
100 REM display menu
110 PRINT "1.  Routine 1"
120 PRINT "2.  Routine 2"
130 PRINT "3.  Routine 3"
140 PRINT "4.  Routine 4"
150 INPUT "Your choice?"; CHOICE
160 ON CHOICE GOSUB 200, 300, 400, 500
170 GOTO 100
180 ' redisplay menu after routine is done
200 REM start of first routine
.
.
.
290 RETURN
300 REM start of second routine
.
.
.
```

ON KEY(n) Statement

Purpose: Sets up a line number for BASIC to trap to when the specified function key or cursor control key is pressed.

Versions:	Cassette	Disk	Advanced	Compiler
			***	(**)

Format: ON KEY(*n*) GOSUB *line*

Remarks: *n* is a numeric expression in the range 1 to 20 indicating the key to be trapped, as follows:

1-10	function keys F1 to F10
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down
15-20	keys defined by the form: KEY *n*,CHR$(*shift*)+CHR$(*scan code*). See "KEY(n) Statement" in this chapter for more information. Keys 15-20 can be trapped in BASIC release 2.0 only.

line is the line number of the beginning of the trapping routine for the specified key. Setting *line* equal to 0 (zero) stops trapping of the key.

A KEY(*n*) ON statement must be executed to activate this statement. After KEY(*n*) ON, if a non-zero line number is specified in the ON KEY(*n*) statement then every time the program starts a new statement, BASIC checks to see if the specified key was pressed. If so, BASIC performs a GOSUB to the specified *line*.

If a KEY(*n*) OFF statement is executed, no trapping takes place for the specified key. Even if the key is pressed, the event is not remembered.

If a KEY(*n*) STOP statement is executed, no trapping takes place for the specified key. However, if the key is pressed the event is remembered, so an immediate trap takes place when KEY(*n*) ON is executed.

When the trap occurs an automatic KEY(*n*) STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a KEY(*n*) ON unless an explicit KEY(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

Key trapping may not work when other keys are pressed before the specified key. The key that caused the trap cannot be tested using INPUT$ or INKEY$, so the trap routine for each key must be different if a different function is desired.

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

KEY(*n*) ON has no effect on whether the soft key values are displayed at the bottom of the screen.

Special considerations for ON KEY statements with DOS national diskettes.

The DOS national diskette keyboard programs have a feature which allows you to change between the United States and national keyboard at any time. You use the F1 or F2 key while holding down Alt and Ctrl to perform the switch. (See your DOS 2.0 manual for more information on the DOS keyboard programs). If your BASIC program traps either of these keys, it will not pass the information to the DOS keyboard program and the keyboard change will not take place. If your program needs to provide this ability to change keyboard formats, you should avoid trapping the F1 and F2 keys.

> **Note:** The shift state you use when trapping either of these keys makes no difference when considering the DOS keyboard programs. Any shift or base state trapping of F1 and F2 will prevent the keystroke from being passed to the DOS program.

Example: The following is an example of a trap routine for function key 5.

```
100 ON KEY(5) GOSUB 200
110 KEY(5) ON
 .
 .
 .
200 REM function key 5 pressed
 .
 .
 .
290 RETURN 140
```

This example traps Ctrl-Break and Ctrl-Alt-Del. Note that this example works in BASIC release 2.0 only.

```
10 KEY 15,CHR$(&H04)+CHR$(70) 'Trap Ctrl-Break
20 KEY 16,CHR$(&H04+&H08)+CHR$(83) 'Trap Ctrl-Alt-Del
30 ON KEY (15) GOSUB 1000
40 ON KEY (16) GOSUB 2000
50 KEY (15) ON: KEY (16) ON
          .
          .
          .
1000 PRINT "Trapping for Ctrl-Break"
1010 RETURN
2000 TRAPS=TRAPS+1
2010 ON TRAPS GOTO 2100,2200,2300,2400,2500
2020 '
2100 PRINT "First trap of System Reset":RETURN
2200 PRINT "Second trap of System Reset":RETURN
2300 PRINT "Third trap of System Reset":RETURN
2400 PRINT "Fourth trap of System Reset":RETURN
2500 KEY (16) OFF 'Disable trapping of System Reset
2510 RETURN
```

NOTES

ON PLAY (n) Statement

Purpose:	Allows continuous background music to play during program execution. For use in BASIC release 2.0 only.

Versions:	Cassette	Disk	Advanced	Compiler
			***	

Format: ON PLAY(*n*) GOSUB *line*

Remarks:

n	is an integer expression in the range 1 to 32 indicating the notes to be trapped. Values entered outside of this range result in an Illegal function call error.
line	is the beginning line number of the trap routine for PLAY. A line number of 0 (zero) stops play trapping.

A PLAY ON statement must be used to start the ON PLAY(n) statement. After PLAY ON, if a non-zero line number is specified in the PLAY(*n*) statement, each time the program starts a new statement BASIC checks to see if the music buffer has gone from *n* go *n*−1 notes. If so, BASIC performs a GOSUB to the specified line.

If PLAY OFF is used, no trapping takes place. Even if a play activity takes place, the event is not remembered.

If a PLAY STOP statement is used, no trapping takes place, but play activity is remembered so that an immediate trap takes place when PLAY ON is executed.

When the trap occurs, an automatic PLAY STOP is run so recursive traps can never take place. The RETURN from the trap routine automatically does a PLAY ON unless an explicit PLAY OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not running a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

You can use RETURN *line* if you want to go back to the BASIC program at a fixed line number. You must use this non-local return with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

Notes:

- A PLAY event trap is only issued when PLAY is in the Music Background mode (PLAY "MB..."). An event trap is not issued when PLAY is in the Music Foreground mode (PLAY "MF...").
- A PLAY event trap is not issued if the Music Background buffer is already empty when a PLAY ON statement is performed.
- Be careful choosing values for *n*. For example: ON PLAY(32) causes so many event traps that little time remains to run the rest of the program.

ON PLAY (n)
Statement

Refer to the PLAY(n) function in this chapter for additional information.

Example: This example sets up a trap routine which is invoked when five notes are left in the background music buffer.

```
10 ON PLAY(5) GOSUB 500
20 PLAY ON
  .
  .
  .
500 REM subroutine for background music
  .
  .
  .
650 RETURN 30
```

NOTES

ON TIMER Statement

Purpose:	Transfers control to a given line number in a BASIC program when a defined period of time has elapsed.

Versions:	Cassette	Disk	Advanced	Compiler
			***	

Format: ON TIMER(*n*) GOSUB *line*

Remarks:

n	is a numeric expression in the range 1 to 86,400 (1 second through 24 hours). Values entered that are outside of this range result in an Illegal function call error.
line	is the beginning line number of the trap routine for TIMER. A line number of 0 (zero) stops timer trapping.

A TIMER ON statement must be used to start the ON TIMER statement. After TIMER ON, specifying a non-zero line number in the ON TIMER(*n*) statement causes BASIC to keep track of the passing seconds. When *n* seconds have elapsed, BASIC performs a GOSUB to the specified line. The event trap occurs, and BASIC starts counting again from 0.

If TIMER OFF is used, no trapping takes place. Even if TIMER activity takes place, the event is not remembered.

If a TIMER STOP statement is used, no trapping takes place, but TIMER activity is remembered so that an immediate trap occurs when TIMER ON is used.

When the trap occurs, an automatic TIMER STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a TIMER ON unless an explicit TIMER OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not running a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), KEY(n) and PLAY).

You can use RETURN *line* if you want to go back to the BASIC program at a fixed line number. You must use this non-local return with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

ON TIMER is useful in programs that need an interval timer. For example, to display the time of day on line one every minute:

```
10 ON TIMER(60) GOSUB 10000
20 TIMER ON
.
.
.
10000 OLDROW=CSRLIN 'save current row
10010 OLDCOL=POS(0) 'save current column
10020 LOCATE 1,1:PRINT TIME$;
10030 LOCATE OLDROW,OLDCOL 'restore row and column
10040 RETURN
```

OPEN Statement

Purpose: Allows I/O to a file or device.

Versions:	Cassette	Disk	Advanced	Compiler
	***	***	***	(**)

Format: First form:

OPEN *filespec* [FOR *mode*] AS [#]*filenum* [LEN=*recl*]

OPEN *path* [FOR *mode*] AS [#]*filenum* [LEN=*recl*]

Alternate form:

OPEN *mode2*, [#]*filenum, filespec* [,*recl*]

OPEN *mode 2* [#]*filenum,path* [,*recl*]

Remarks: *mode* in the first form, is one of the following:

OUTPUT	specifies sequential output mode.
INPUT	specifies sequential input mode.
APPEND	specifies sequential output mode where the file is positioned to the end of data on the file when it is opened.

Note that *mode* must be a string constant, *not* enclosed in quotation marks. If *mode* is omitted, random access is assumed.

mode2	in the alternate form, is a string expression with the first character being one of the following:
	O specifies sequential output mode
	I specifies sequential input mode
	R specifies random input/output mode

For both formats:

filenum	is an integer expression whose value is between one and the maximum number of files allowed. In Cassette BASIC, the maximum number is 4. In Disk and Advanced BASIC, the default maximum is 3, but this can be changed with the /F: switch on the BASIC command.
filespec	is a string expression for the file specification as explained under "Naming Files" in Chapter 3.
path	is a string expression not exceeding 63 characters as explained under "Naming Files" in Chapter 3. Refer also to "Tree-Structured Directories" in Chapter 3.

recl	is an integer expression which, if included, sets the record length for random files. It may range from 1 to 32767. In BASIC release 1.1 or earlier, *recl* is not valid for sequential files. However, in BASIC release 2.0, you can use *recl* for sequential files. The default record length is 128 bytes. *recl* may not exceed the value set by the **/S:** switch on the BASIC command. You can also use LEN= for sequential files.

OPEN allocates a buffer for I/O to the file or device and determines the mode of access that will be used with the buffer.

filenum is the number that is associated with the file for as long as it is open and is used by other I/O statements to refer to the file or device.

An OPEN must be executed before any I/O may be done to a device or file using any of the following statements, or any statement or function requiring a file number:

```
PRINT #              WRITE #
PRINT # USING        INPUT$
INPUT #              GET
LINE INPUT #         PUT
```

GET and PUT are valid for random files (or communications files – see the "OPEN "COM... Statement" section). A diskette file may be either random or sequential, and a printer may be opened in either random or sequential mode; however, all other devices may be opened only for sequential operations.

BASIC normally adds a line feed after each carriage return (CHR$(13)) sent to a printer. However, if you open a printer (LPT1, LPT2, or LPT3) as a random file with width 255, this line feed is suppressed.

APPEND is valid only for diskette files. The file pointer is initially set to the end of the file and the record number is set to the last record of the file. PRINT # or WRITE # will then extend the file.

> **Note:** At any one time, it is possible to have a particular file open under more than one file number. This allows different modes to be used for different purposes. Or, for program clarity, you may use different file numbers for different modes of access. Each file number has a different buffer, so you should use care if you are writing using one file number and reading using another file number.

However, a file cannot be opened for sequential output or append if the file is already open.

If the device name is omitted when you are using Cassette BASIC, CAS1 is assumed. If you are using Disk or Advanced BASIC, the DOS default drive is assumed.

If CAS1 is specified as the device and the filename is omitted, then the next data file on the cassette is opened.

In Cassette BASIC, a maximum of four files may be open at one time (cassette, printer, keyboard, and screen). Note that only one cassette file may be open at a time. For Disk and Advanced BASIC the default maximum is three files. You can override this value by using the /F: option on the BASIC command.

If a file opened for input does not exist, a **File not found** error occurs. If a file which does not exist is opened for output, append, or random access, a file is created.

Any values given outside the ranges indicated will result in an **Illegal function call** error. The file is not opened.

See "Appendix B. BASIC Diskette Input and Output" for a complete explanation of using diskette files. Refer to the next section, "OPEN "COM... Statement," for information on opening communications files.

OPEN Statement

Examples: Either of these statements opens the file named "DATA" for sequential output on the default device (CAS1 for Cassette BASIC, default drive for Disk and Advanced BASIC).

```
10 OPEN "DATA" FOR OUTPUT AS #1
```

or

```
10 OPEN "O",#1,"DATA"
```

In the above example, note that the opening for output destroys any existing data in the file. If you do not wish to destroy data you should open for APPEND.

```
20 OPEN "B:SSFILE" AS 1 LEN=256
```

or

```
20 OPEN "R",1,"B:SSFILE",256
```

Either of the preceding two statements opens the file named "SSFILE" on the diskette in drive B for random input and output. The record length is 256.

```
25 FILE$ = "A :DATA.ART"
30 OPEN FILE$ FOR APPEND AS 3
```

This example opens the file "DATA.ART" on the diskette in drive A and positions the file pointers so that any output to the file is placed at the end of existing data in the file.

```
Ok
  1Ø OPEN "LPT1:" AS #1' random access
  2Ø PRINT #1,"Printing width 8Ø"
  3Ø PRINT #1,"Now change to width 255"
  4Ø WIDTH #1,255
  5Ø PRINT #1,"This line will be underlined"
  6Ø WIDTH #1,8Ø
  7Ø PRINT #1, STRING$(28,"_")
  8Ø PRINT #1,"Printing width 8Ø with CR/LF"
RUN
OK
```

This is printed on the printer:

```
Printing width 80
Now change to width 255
This line will be underlined
Printing width 80 with CR/LF
```

Line 10 in this example opens the printer in random mode. Because the default width is 80, the lines printed by lines 20 and 30 end with a carriage return/line feed. Line 40 changes the printer width to 255, so the line feed after the carriage return is suppressed. Therefore, the line printed by line 50 ends only with a carriage return and not a line feed. This causes the line printed by line 70 to overprint "This line will be underlined", causing the line to be underlined. Line 60 changes the width back to 80 so the underlines and following lines will end with a line feed.

OPEN Statement

In BASIC 2.0 it is possible to OPEN files using paths as described under "Naming Files" in Chapter 3. The following examples illustrate the use of paths for filespec.

```
10 OPEN "LVL1\LVL2\DATA" FOR OUTPUT AS #1
```

or

```
10 OPEN "O",#1,"LVL1\LVL2\DATA"
```

Either of these statements opens the file called "DATA" for sequential output on the default device in the directory called LVL2.

```
20 OPEN "B:LVL1\RRFILE" AS 1 LEN=256
```

or

```
20 OPEN "R",1,"B:LVL1\RRFILE",256
```

Either of the preceding two statements opens the file named "RRFILE" in the LVL1 directory on the diskette in drive B for random input and output. The record length is 256.

```
25 FILE$="A:LVL1\LVL2\LVL3\DATA.FIL"
30 OPEN FILE$ FOR APPEND AS 3
```

This example opens the file "DATA.FIL" on the diskette in drive A in the directory called LVL3 and positions the file pointers so that any output to the file is placed at the end of the existing data in the file.

```
10 CHDIR "LVL1"
20 OPEN "..\LVL2\PROG" FOR OUTPUT AS #1 Len=32
```

In this example, the file PROG is opened for output in the LVL2 directory.

Note: In BASIC 2.0, it is possible to open a sequential file and set the record length.

NOTES

STATEMENTS

OPEN "COM... Statement

Purpose:	Opens a communications file.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	(**)

Valid only with Asynchronous Communications Adapter.

Format: OPEN "COM*n*:[*speed*] [,*parity*] [,*data*] [,*stop*] [,RS] [,CS[*n*]] [,DS[*n*]] [,CD[*n*]] [,LF] [,PE]" AS [#]*filenum* [LEN=*number*]

Remarks:

n	is 1 or 2, indicating the number of the Asynchronous Communications Adapter.
speed	is an integer constant specifying the transmit/receive bit rate in bits per second (bps). Valid speeds are 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, and 9600. The default is 300 bps.

parity is a one-character constant specifying the parity for transmit and receive as follows:

S	SPACE: Parity bit always transmitted and received as a space (0 bit).
O	ODD: Odd transmit parity, odd receive parity checking.
M	MARK: Parity bit always transmitted and received as a mark (1 bit).
E	EVEN: Even transmit parity, even receive parity checking.
N	NONE: No transmit parity, no receive parity checking.

The default is EVEN (E).

data is an integer constant indicating the number of transmit/receive data bits. Valid values are: 4, 5, 6, 7, or 8. The default is 7.

stop is an integer constant indicating the number of stop bits. Valid values are 1 or 2. The default is two stop bits for 75 and 110 bps, one stop bit for all others. If you use 4 or 5 for *data*, a 2 here will mean 1½ stop bits.

OPEN "COM... Statement

filenum	is an integer expression which evalutates to a valid file number. The number is then associated with the file for as long as it is open and is used by other communications I/O statements to refer to the file.
number	is the maximum number of bytes which can be read from the communication buffer when using GET or PUT. The default is 128 bytes.

OPEN "COM... allocates a buffer for I/O in the same fashion as OPEN for diskette files. It supports RS232 asynchronous communication with other computers and peripherals.

A communications device may be open to only one file number at a time.

The RS, CS, DS, CD, LF and PE options affect the line signals as follows:

RS	suppresses RTS (Request To Send)
CS[n]	controls CTS (Clear To Send)
DS[n]	controls DSR (Data Set Ready)
CD[n]	controls CD (Carrier Detect)
LF	sends a line feed following each carriage return
PE	enables parity checking

The CD (Carrier Detect) is also known as the RLSD (Received Line Signal Detect).

> **Note:** The *speed, parity, data*, and stop parameters are positional, but **RS, CS, DS, CD, LF,** and **PE** are not.

The RTS (Request To Send) line is turned on when you execute an OPEN "COM... statement unless you include the **RS** option.

The *n* argument in the **CS, DS,** and **CD** options specifies the number of milliseconds to wait for the signal before returning a **Device Timeout** error. *n* may range from 0 to 65535. If *n* is omitted or is equal to zero, then the line status is not checked at all.

The defaults are **CS1000, DS1000,** and **CD0**. If **RS** was specified, **CS0** is the default.

That is, normally I/O statements to a communications file will fail if the CTS (Clear To Send) or DSR (Data Set Ready) signals are off. The system waits one second before returning a **Device Timeout**. The **CS** and **DS** options allow you to ignore these lines or to specify the amount of time to wait before the timeout.

Normally Carrier Detect (CD or RLSD) is ignored when an OPEN "COM... statement is executed. The **CD** option allows you to test this line by including the *n* parameter, in the same way as **CS** and **DS**. If *n* is omitted or is equal to zero, then Carrier Detect is not checked at all (which is the same as omitting the **CD** option).

The **LF** parameter is intended for those using communication files as a means of printing to a serial line printer. When you specify **LF**, a line feed character (hex 0A) is automatically sent after each carriage return character (hex 0C). (This includes the carriage return sent as a result of the width setting.) Note that INPUT # and LINE INPUT #, when used to read from a communications file that was opened with the **LF** option, stop when they see a carriage return. The line feed is always ignored.

The **PE** option enables parity checking. The default is no parity checking. The **PE** option will cause a **Device I/O** error on parity errors and will turn the high order bit on for 7 or less data bits. The **PE** option does *not* affect framing and overrun errors. These errors will always turn on the high order bit and cause a **Device I/O** error.

Any coding errors within the string expression starting with *speed* result in a **Bad file name** error. An indication as to which parameter is in error is not given.

Refer to "Appendix F. Communications" for more information on control of output signals and other technical information on communications support.

If you specify 8 data bits, you must specify parity N. If you specify 4 data bits, you must specify a parity, that is, N parity is invalid. BASIC uses all 8 bits in a byte to store numbers, so if you are transmitting or receiving numeric data (for example, by using PUT), you must specify 8 data bits. (This is not so if you are sending numeric data *as text*.)

Refer to the previous section for opening devices other than communications devices.

Examples:

```
10 OPEN "COM1:" AS 1
```

File 1 is opened for communication with all defaults. The speed is 300 bps with even parity. There will be 7 data bits and one stop bit.

```
10 OPEN "COM1:2400" AS #2
```

File 2 is opened for communication at 2400 bps. Parity, number of data bits, and number of stop bits are defaulted.

```
20 OPEN "COM2:1200,N,8" AS #1
```

File number 1 is opened for asynchronous I/O at 1200 bps, no parity is to be produced or checked, 8-bit bytes will be sent and received, and 1 stop bit will be transmitted.

```
10 OPEN "COM1:9600,N,8,,CS,DS,CD" AS #1
```

Opens COM1 at 9600 bps with no parity and eight data bits. CTS, DSR, and RLSD are not checked.

```
50 OPEN "COM1:1200,,,,CS,DS2000" AS #1
```

Opens COM1 at 1200 bps with the defaults of even parity and seven data bits. RTS is sent, CTS is not checked, and **Device Timeout** is given if DSR is not seen within two seconds. Note that the commas are required to indicate the position of the *parity*, *start*, and *stop* parameters, even though a value is not specified. This is what is meant by *positional* parameters.

An OPEN statement may be used with an ON ERROR statement to make sure a modem is working properly before sending any data. For example, the following program makes sure we get Carrier Detect (CD or RLSD) from the modem before starting. Line 20 is set to timeout after 10 seconds. TRIES is set to 6 so we give up if Carrier Detect is not seen within one minute. Once communication is established, we reopen the file with a shorter delay until timeout.

```
5 TRIES=6
10 ON ERROR GOTO 100
20 OPEN "COM1:300,N,8,2,CS,DS,CD10000" AS #1
30 ON ERROR GOTO 0
40 CLOSE #1 ' works so can continue
50 GOTO 1000
 .
 .
 .
100/TRIES=TRIES-1
110 IF TRIES=0 THEN ON ERROR GOTO 0 ' give up
120 RESUME
 .
 .
 .
1000 OPEN "COM1:300,N,8,2,CS,DS,CD2000" AS #1
```

The next example shows a typical way to use a communication file to control a serial line printer. The LF parameter in the OPEN statement ensures that lines do not print on top of each other.

```
10 WIDTH "COM1:", 132
20 OPEN "COM1:1200,N,8,,CS10000,DS10000,CD10000,LF"
      AS #1
```

OPTION BASE Statement

Purpose: Declares the minimum value for array subscripts.

Versions:	Cassette	Disk	Advanced	Compiler
	***	***	***	***

Format: OPTION BASE *n*

Remarks: *n* is 1 or 0.

The default base is 0. If the statement:

```
OPTION
```

is executed, the lowest value an array subscript may have is one.

The OPTION BASE statement must be coded *before* you define or use any arrays. An error occurs if you change the base value when arrays exist.

In BASIC 1.10 or earlier, the program you are chaining to cannot have an OPTION BASE statement even when both programs have the same base value.

Purpose: Fills in an area on the screen with the selected color.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Graphics mode only.

Format: PAINT *(x,y)* [[*,paint*] [*,boundary*] [*,background*]]

Remarks:

(x,y) are the coordinates of a point within the area to be filled in. The coordinates may be given in absolute or relative form (see "Specifying Coordinates" under "Graphics Modes" in Chapter 3). This point will be used as a starting point.

paint can be a numeric or string exression. If *paint* is a numeric expression, it is the color to be painted with, in the range 0 to 3. In medium resolution, this color is the color from the current palette as defined by the COLOR statement. Zero (0) is the background color. The default is the foreground color, color number 3. In high resolution, *paint* equal to 0 (zero) indicates black, and the default of 1 (one) indicates white.

If *paint* is a string expression, then "tiling" is performed, as described later in this section. Paint tiling is for use in BASIC release 2.0 only.

boundary	is the color of the edges of the figure to be filled in, in the range 0 to 3 as described above.
background	is a one-byte string expression used in paint tiling. (For use in BASIC release 2.0 only.)

The figure to be filled in is the figure with edges of *boundary* color. That is, the figure should be drawn in the *boundary* color. The figure is filled in with the color *paint*.

Since there are only two colors in high resolution, *paint* should not be different from *boundary*. Since *boundary* is defaulted to equal *paint* we don't need the third parameter in high resolution mode.

In high resolution this means "blacking out" an area until black is encountered, or "whiting out" an area until white is encountered.

In medium resolution, we can fill in a border of color 2 with color 1. Visually this might mean a green ball with a red border.

The starting point of PAINT must be inside the figure to be painted. Points plotted that are outside of the screen limits are not drawn and no error occurs. If the specified point already has the color *boundary*, the PAINT will have no effect. If *paint* is omitted the foreground color is used (3 in medium resolution, 1 in high resolution). PAINT can paint any type of figure; "jagged" edges on a figure will increase the amount of stack space required by PAINT. So if a lot of complex painting is being done you may want to use CLEAR at the beginning of the program to increase the stack space available.

The PAINT statement allows scenes to be displayed with very few statements.

The PAINT statement in line 20 fills in the box drawn in line 10 with color number 1.

```
5 SCREEN 1
1Ø LINE (Ø,Ø)-(1ØØ,15Ø),2,B
2Ø PAINT (5Ø,5Ø),1,2
```

The following discussion deals with paint tiling only, and is for use in BASIC release 2.0 only.

To use paint tiling, the *paint* attribute must be a string expression of the form.

CHR$(&Hnn)+CHR$(&Hnn)+CHR$(&Hnn)

The tile mask is always 8 bits wide. The two hexadecimal numbers in the CHR$ expression correspond to 8 bits. The string expression may contain up to 64 bytes. The structure of the string expression appears as follows:

```
             x increases -->
            bit of tile byte
x,y          8 7 6 5 4 3 2 1
Ø,Ø          x x x x x x x x    Tile byte Ø
Ø,1          x x x x x x x x    Tile byte 1
Ø,2          x x x x x x x x    Tile byte 2
.
.
.
Ø,63         x x x x x x x x    Tile byte 63 (maximum allowed)
```

The tile pattern is repeated uniformly over the entire screen. Each byte in the tile string masks 8 bits along the x axis when plotting points. Each byte in the tile string is rotated as required to align the y axis, such that tile_byte_mask=y mod tile_length.

Because there is only one bit per pixel in high resolution, a point is plotted at every position in the bit mask which has a value of 1. In high resolution, the screen can be painted with x's using the following example:

```
PAINT (32Ø,1ØØ), CHR$(&H81)+CHR$(&H42)+
  CHR$(&H24)+CHR$(&H18)+CHR$(&H18)+
  CHR$(&H24)+CHR$(&H42)+CHR$(&H81)
```

The length of this mask is eight, indexed zero through seven. In this case, PAINT at coordinates (320,100) will begin by plotting byte four. This is calculated using the y **mod** tile__length formula by substituting 100 for y and eight for tile__length. This pattern appears on the screen as:

```
                  x increases -->
Tile byte Ø     1 Ø Ø Ø Ø Ø Ø 1   CHR$(&H81)
Tile byte 1     Ø 1 Ø Ø Ø Ø 1 Ø   CHR$(&H42)
Tile byte 2     Ø Ø 1 Ø Ø 1 Ø Ø   CHR$(&H24)
Tile byte 3     Ø Ø Ø 1 1 Ø Ø Ø   CHR$(&H18)
Tile byte 4     Ø Ø Ø 1 1 Ø Ø Ø   CHR$(&H18)
Tile byte 5     Ø Ø 1 Ø Ø 1 Ø Ø   CHR$(&H24)
Tile byte 6     Ø 1 Ø Ø Ø Ø 1 Ø   CHR$(&H42)
Tile byte 7     1 Ø Ø Ø Ø Ø Ø 1   CHR$(&H81)
```

The method of designing patterns in medium resolution is different from the method used in high resolution. In high resolution, each byte of the string is able to plot eight points across the screen (one bit per pixel).

However, one medium resolution tile byte describes only four pixels since medium resolution has only two bits per pixel. Every two bits of the tile byte describes one of four possible colors associated with each of the four pixels to be plotted.

The following chart shows the binary and hexadecimal values associated with the given colors.

Color palette 0	Color palette 1	Color number in binary	Pattern to draw solid line in binary	Pattern to draw solid line in hexadecimal
green	cyan	01	01010101	&H55
red	magenta	10	10101010	&HAA
brown	white	11	11111111	&HFF

In medium resolution, the following example plots a pattern of boxes with a border color of red in palette 0 and magenta in palette 1.

```
PAINT (32Ø,1ØØ),CHR$(&HAA)+CHR$(&H82)+CHR$(&H82)
+CHR$(&H82)+CHR$(&H82)+CHR$(&H82)+CHR$(&H82)
+CHR$(&HAA)

              7 6 5 4 3 2 1 Ø

Tile byte Ø   1 Ø 1 Ø 1 Ø 1 Ø   CHR$(&HAA)
Tile byte 1   1 Ø Ø Ø Ø Ø 1 Ø   CHR$(&H82)
Tile byte 2   1 Ø Ø Ø Ø Ø 1 Ø   CHR$(&H82)
Tile byte 3   1 Ø Ø Ø Ø Ø 1 Ø   CHR$(&H82)
Tile byte 4   1 Ø Ø Ø Ø Ø 1 Ø   CHR$(&H82)
Tile byte 5   1 Ø Ø Ø Ø Ø 1 Ø   CHR$(&H82)
Tile byte 6   1 Ø Ø Ø Ø Ø 1 Ø   CHR$(&H82)
Tile byte 7   1 Ø 1 Ø 1 Ø 1 Ø   CHR$(&HAA)
```

Occasionally, you may want to tile paint over an already painted area that is the same color as two consecutive lines in the tile pattern. Normally, this constitutes a terminating condition because your point is surrounded by the same color.

You can use the *background* attribute to skip this terminating condition. You cannot specify more than two consecutive lines in the tile pattern matching this *background* attribute. Specifying more than two consecutive lines in the tile pattern that match *background* causes an **Illegal function call** error.

Examples: The program below will demonstrate how to tile an area with three lines of red, two lines of green, and one line of red. The palette will then change to show that the same tile mask will yield the same pattern with different colors.

```
10 CLS:SCREEN 1,0:KEY OFF
20 TIL$=CHR$(&HAA)+CHR$(&HAA)+CHR$(&HAA)
   +CHR$(&H55)+CHR$(&H55)+CHR$(&HFF)
30 COLOR 0,0  'choose palette 1
40 VIEW (1,1)-(150,100),0,2
50 GOSUB 1000
60 COLOR 0,1  'choose palette 1
70 GOTO 1020
1000 PAINT (125,50),TIL$,2
1010 RETURN
1020 END
```

The following example uses paint tiling with the *background* attribute.

```
10 SCREEN 1:CLS:COLOR 0,1
20 TIL$=CHR$(&H5F)+CHR$(&H5F)
   +CHR$(&H27)+CHR$(&H81)
30 VIEW (1,1)-(150,100),0,2
40 LOCATE 3,22:PRINT "<---Without back-"
50 LOCATE 4,22:PRINT "    ground tile "
60 PAINT (125,50),CHR$(&H5F)
70 PAINT (125,50),TIL$,2
80 '
90 ' with background tile
100 '
110 VIEW (160,100)-(310,198),0,2
120 LOCATE 16,1:PRINT "With background-->"
130 LOCATE 17,1:PRINT "tile CHR$(&H5F)"
140 PAINT (125,50),CHR$(&H5F)
150 PAINT (125,50),TIL$,2,CHR$(&H5F)
160 LINE (1,100)-(319,100),3
170 FOR I=1 TO 2500:NEXT I
180 END
```

PLAY Statement

Purpose: Plays music as specified by *string*.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: PLAY *string*

Remarks: PLAY implements a concept similar to DRAW by imbedding a "tune definition language" into a character string.

string is a string expression consisting of single character music commands.

The single character commands in PLAY are:

A to G with optional #, +, or −
Plays the indicated note in the current octave. A number sign (#) or plus sign (+) afterwards indicates a sharp, a minus sign (−) indicates a flat. The #, +, or − is not allowed unless it corresponds to a black key on a piano. For example, B# is an invalid note.

O n Octave. Sets the current octave for the following notes. There are 7 octaves, numbered 0 to 6. Each octave goes from C to B. Octave 3 starts with middle C. Octave 4 is the default octave.

> n	Go up to the next higher octave and play note n. Each time note n is played the octave goes up, until it reaches octave 6. For example, PLAY ">A" raises the octave and plays note A. Each time PLAY ">A" is executed the octave goes up until it reaches octave 6, then each time PLAY ">A" executes, note A plays at octave 6. This command is supported only in BASIC release 2.0.
< n	Go down one octave and play note n. Each time note n is played the octave goes down, until it reaches octave 0. For example, PLAY "<A" lowers the octave and plays note A. Each time PLAY "<A" is executed the octave goes down until it reaches octave 0, then each time PLAY "<A" executes, note A plays at octave 0. This command is supported only in BASIC release 2.0.
N n	Plays note n. n may range from 0 to 84. In the 7 possible octaves, there are 84 notes. n=0 means rest. This is an alternative way of selecting notes besides specifying the octave (0 n) and the note name (A-G).

L n Sets the length of the following notes. The actual note length is 1/n. n may range from 1 to 64. The following table may help explain this:

Length	Equivalent
L1	whole note
L2	half note
L3	one of a triplet of three half notes (1/3 of a 4 beat measure)
L4	quarter note
L5	one of a quintuplet (1/5 of a measure)
L6	one of a quarter note triplet
.	
.	
.	
L64	sixty-fourth note

The length may also follow the note when you want to change the length only for the note. For example, A16 is equivalent to L16A.

P n Pause (rest). n may range from 1 to 64, and figures the length of the pause in the same way as L (length).

. (dot or period) After a note, causes the note to be played as a dotted note. That is, its length is multiplied by 3/2. More than one dot may appear after the note, and the length is adjusted accordingly. For example, "A.." will play 9/4 as long as L specifies, "A..." will play 27/8 as long, etc. Dots may also appear after a pause (P) to scale the pause length in the same way.

T n	Tempo. Sets the number of quarter notes in a minute. n may range from 32 to 255. The default is 120. Under "SOUND Statement," later in this chapter, is a table listing common tempos and the equivalent beats per minute.
MF	Music foreground. Music (created by SOUND or PLAY) runs in foreground. That is, each subsequent note or sound will not start until the previous note or sound is finished. You can press Ctrl-Break to exit PLAY. Music foreground is the default state.
MB	Music background. Music (created by SOUND or PLAY) runs in background instead of in foreground. That is, each note or sound is placed in a buffer allowing the BASIC program to continue executing while music plays in the background. Up to 32 notes (or rests) may be played in background at a time.
MN	Music normal. Each note plays 7/8 of the time specified by L (length). This is the default setting of MN, ML, and MS.
ML	Music legato. Each note plays the full period set by L (length).
MS	Music staccato. Each note plays 3/4 of the time specified by L.
X variable;	Executes specified string.

In all these commands the *n* argument can be a constant like 12 or it can be =*variable;* where *variable* is the name of a variable. The semicolon (;) is required when you use a variable in this way, and when you use the **X** command. Otherwise a semicolon is optional between commands, except a semicolon is not allowed after **MF**, **MB**, **MN**, **ML**, or **MS.** Also, any blanks in *string* are ignored.

You can also specify variables in the form VARPTR$(*variable*), instead of =*variable;*. The VARPTR$ form is the only one that can be used in compiled programs. For example:

One Method	**Alternative Method**
`PLAY "XA$;"`	`PLAY "X"+VARPTR$(A$)`
`PLAY "O=I;"`	`PLAY "O="+VARPTR$(I)`

You can use **X** to store a "subtune" in one string and call it repetitively with different tempos or octaves from another string.

Examples: The following example plays a tune.

```
10 REM little lamb
20 MARY$="GFE-FGGG"
30 PLAY "MB T100 O3 L8;XMARY$;P8 FFF4"
40 PLAY "GB-B-4; XMARY$; GFFGFE-."
```

The following example plays the scale from octave 0 to octave 6.

```
10 ' Play the scale using > octave
20 SCALE$="CDEFGAB"
30 PLAY "O0 XSCALE$;"
40 FOR I=1 TO 6
50 PLAY ">XSCALE$;"
60 NEXT
70 ' Play the scale using < octave
80 PLAY "O6 XSCALE$;"
90 FOR I-1 TO 6
100 PLAY "<XSCALE$;"
110 NEXT
```

PLAY(n) Function

Purpose: Returns the number of notes currently in the music background buffer. For use in BASIC release 2.0 only.

Versions:	Cassette	Disk	Advanced	Compiler
			***	

Format: *v*=PLAY(*n*)

Remarks: *n* is a dummy argument that can have any value.

PLAY(*n*) returns a 0 when the program is running in Music Foreground mode. The maximum value that can be returned is 32, which is the maximum number of notes held in the buffer.

PLAY(n) only returns notes in the buffer when you are using Music Background (MB) mode.

Example:

```
10 'when 3 notes are left in
20 'the background music buffer
30 'go to line 1000
40 'and play another tune
50 PLAY "MB CDEFGAB"
60 IF PLAY (0)=5 GOTO 1000
  .
  .
  .
1000 PLAY "MB O4 T200 L4 MS GG#GE"
```

STATEMENTS

PMAP

Purpose: Maps physical coordinates to world coordinates or world coordinates to physical coordinates. For use with graphics in BASIC release 2.0 only.

Versions:	Cassette	Disk	Advanced	Compiler
			***	

Graphics mode only

Format: *v*=PMAP(*x*,*n*)

Remarks: *x* coordinate of the point that is to be mapped

n may be a value in the range 0 to 3 such that

0 maps the world coordinate x to the physical coordinate x

1 maps the world coordinate y to the physical coordinate y

2 maps the physical coordinate x to the world coordinate x

3 maps the physical coordinate y to the world coordinate y

PMAP is used to translate coordinates between the world system as defined by the WINDOW statement and the physical coordinate system as defined by the VIEW statement.

PMAP(x,0) and PMAP(x,1) are used to map values from the world coordinate system to the physical coordinate system.

PMAP(x,2) and PMAP(x,3) are used to map values from the physical coordinate system to the world coordinate system.

For example, if the statement

```
SCREEN 1: WINDOW (-1,-1)-(1,1)
```

is in effect we can use PMAP to map the world coordinate points of (−1,−1) and (1,1) to their corresponding physical points on the screen.

PMAP(−1,0) returns the physical x coordinate value of 0.

PMAP(−1,1) returns the physical y coordinate value of 199.

PMAP(1,0) returns the physical x coordinate value of 319.

PMAP(1,1) returns the physical y coordinate value of 0.

The above information tells us that the point (−1,−1) which is in the lower left corner of the screen corresponds to the physical point (0,199). We also know that the point (1,1) which is in the upper right corner corresponds to the physical point (319,0).

With the statement WINDOW (−1,−1)−(1,1) the center point of the window is (0,0). PMAP(1,0) returns a physical x coordinate of 160. PMAP(0,1) returns a physical y coordinate of 100. So the point (0,0) which is in the center corresponds to the physical point (160,100).

Purpose: Returns the color of the specified point on the screen or current graphics coordinate.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	(**)

Graphics mode only.

Format: *v* = POINT (*x*,*y*)

v = POINT (*n*)

Remarks: (*x*,*y*) are the coordinates of the point to be used. The coordinates must be in absolute form (see "Specifying Coordinates" under "Graphics Modes" in Chapter 3).

If the point given is out of range the value −1 is returned. In medium resolution valid returns are 0, 1, 2, and 3. In high resolution they are 0 and 1.

n	returns the value of the current x or y graphics coordinate. This form is supported only in BASIC release 2.0. *n* can have a value from 0 to 3 where:

0	returns the current physical x coordinate.
1	returns the current physical y coordinate.
2	returns the current world x coordinate if WINDOW is active. If WINDOW is not active, returns the current physical x coordinate.
3	returns the current world y coordinate if WINDOW is active. If WINDOW is not active, returns the current physical y coordinate.

For more information, see "WINDOW Statement" in this chapter.

Examples: The following example inverts the current setting of point (I,I).

```
5 SCREEN 2
1Ø IF POINT (I,I)<>Ø THEN PRESET(I,I)
                      ELSE PSET(I,I)
```

or

```
1Ø PSET(I,I),1-POINT(I,I)
```

The following example illustrates values returned by the POINT function. Note the change in the values depending upon WINDOW.

```
1Ø CLS:SCREEN 1,Ø
15 PRINT "POINT(n) with WINDOW inactive"
2Ø GOSUB 1ØØ
3Ø WINDOW (Ø,Ø)-(319,199)
4Ø PRINT "POINT(n) with WINDOW active"
5Ø GOSUB 1ØØ
6Ø PRINT "POINT(n) with WINDOW and SCREEN active"
7Ø WINDOW SCREEN (Ø,Ø)-(319,199)
8Ø GOSUB 1ØØ
9Ø END
1ØØ PSET (5,15)
11Ø FOR I=Ø TO 3
12Ø PRINT POINT (I);
13Ø NEXT
135 PRINT:PRINT
14Ø RETURN
Ok
RUN

POINT(n) with WINDOW inactive
5  15  5  15
POINT(n) with WINDOW active
5  184  5  15
POINT(n) with WINDOW and SCREEN active
5  15  5  15
```

POKE Statement

Purpose: Writes a byte into a memory location.

Versions:	Cassette	Disk	Advanced	Compiler
	***	***	***	***

Format: POKE *n,m*

Remarks: *n* must be in the range 0 to 65535 and indicates the address of the memory location where the data is to be written. It is an offset from the current segment as defined by the DEF SEG statement (see "DEF SEG Statement" in this chapter).

m m is the data to be written to the specified location. It must be in the range 0 to 255.

The complimentary function to POKE is PEEK. (See "PEEK Function" in this chapter.) POKE and PEEK are useful for efficient data storage, loading machine language subroutines, and passing arguments and results to and from machine language subroutines.

Warning:
BASIC does not do any checking on the address. So don't go POKEing around in BASIC's stack, BASIC's variable area, or your BASIC program.

Example:

```
10 DEF SEG: POKE 106,0
```

See "INKEY$ Variable" in this chapter for an explanation of this example.

Movement done this way leaves the background unchanged. Flicker can be reduced by minimizing the time between steps 4 and 1, and making sure there is enough time delay between steps 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET action verb. But you should remember to have an image area that will contain the "before" and "after" images of the object. This way the extra area will effectively erase the old image. This method may be somewhat faster than the method using XOR described above, since only one PUT is required to move an object (although you must PUT a larger image).

If the image to be transferred is too large to fit on the screen, an **Illegal function call** error occurs.

RANDOMIZE Statement

Purpose:	Reseeds the random number generator.

Versions:	Cassette	Disk	Advanced	Compiler
	***	***	***	(**)

Format: RANDOMIZE [*n*]

RANDOMIZE TIMER

Remarks: *n* is an integer, single- or double-precision expression that is used as the random number seed. In Cassette BASIC, *n* must be an integer expression.

If *n* is omitted, BASIC suspends program execution and asks for a value by displaying:

```
Random Number Seed (-32768 to 32767)?
```

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the seed with each run.

In Disk and Advanced BASIC, the internal clock can be a useful way to get a random number seed. You can use VAL to change the last two digits of TIME$ to a number, and use that number for *n*.

In BASIC release 2.0, you can get a new random number seed without being prompted. To do this, use the TIMER function in the expression. Note that the sequence is different each time the program runs.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 4
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)?
```

Suppose you respond with 3. The program continues:

```
Random Number seed (-32768 to 32767)? 3
 .7655695  .3558607  .3742327  .1388798
Ok
RUN
Random Number Seed (-32768 to 32767)?
```

Suppose this time you respond with 4. The program continues:

```
Random Number Seed (-32768 to 32767)? 4
 .1719568  .5273236  .6879686  .713297
Ok
RUN
Random Number Seed (-32768 to 32767)?
```

If you try 3 again, you'll get the same sequence as the first run:

```
Random Number Seed (-32768 to 32767)? 3
 .7655695  .3558607  .3742327  .1388798
Ok
```

In the program below, note that each time the program is run you see a different sequence of numbers.

```
5 'use of TIMER with basic 2.0
1Ø RANDOMIZE TIMER
2Ø FOR I=1 TO 4
3Ø PRINT RND;
4Ø NEXT
RUN
 .9590051  .1036786  .1464037  .7754918
Ok
RUN
 .8261163  .17422  .9191545  .5041142
Ok
```

NOTES

STATEMENTS

READ Statement

Purpose: Reads values from a DATA statement and assigns them to variables (see "DATA Statement" in this chapter).

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: READ *variable* [,*variable*]...

Remarks:

variable	is a numeric or string variable or array element which is to receive the value read from the DATA table.

A READ statement must always be used with a DATA statement. READ statements assign DATA statement values to the variables in the READ statement on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a **Syntax error** results.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in the list of variables exceeds the number of elements in the DATA statement(s), an **Out of data** error occurs. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

RMDIR Command

Purpose: Removes a directory from the specified diskette. For use in BASIC release 2.0 only.

Versions:

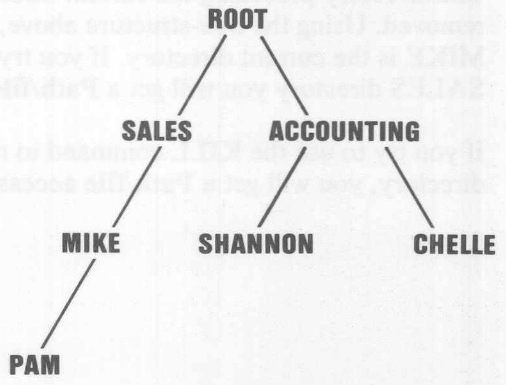
Cassette	Disk	Advanced	Compiler
	***	***	

Format: RMDIR *path*

Remarks: *path* is a string expression, not exceeding 63 characters, that identifies the sub-directory to be removed from the existing directory. Refer to "Naming Files" and "Tree-Structured Directories" in Chapter 3 for more information.

The directory must be empty of all files and sub-directories before it can be removed, with the exception of the '.' and '..' entries, or a Path/file access error occurs.

Examples:



The following examples refer to the tree-structure above.

If you are in the root directory and you want to remove the directory called PAM, use:

```
RMDIR "SALES\MIKE\PAM"
```

If you want to make ACCOUNTING the current directory and remove the directory called CHELLE, use:

```
CHDIR "ACCOUNTING"
RMDIR "CHELLE"
```

Another way to remove the directory CHELLE is to make the root the current directory and then remove CHELLE.

```
CHDIR "\"
RMDIR "ACCOUNTING\CHELLE"
```

The directory preceding the current directory cannot be removed. Using the tree-structure above, suppose that MIKE is the current directory. If you try to remove the SALES directory you will get a **Path/file access** error.

If you try to use the KILL command to remove a directory, you will get a **Path/file access** error.

Purpose: Sets the screen attributes to be used by subsequent statements.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Meaningful with the Color/Graphics Monitor Adapter only.

Format: SCREEN [*mode*] [,[*burst*] [,[*apage*] [,*vpage*]]]

Remarks: *mode* is a numeric expression resulting in an integer value of 0, 1 or 2. Valid modes are:

0	Text mode at current width (40 or 80).
1	Medium resolution graphics mode (320x200). Use with Color/Graphics Monitor Adapter only.
2	High resolution graphics mode (640x200). Use with Color/Graphics Monitor Adapter only.

burst is a numeric expression resulting in a true or false value. It enables color. In text mode (*mode*=0), a false (zero) value disables color (black and white images only) and a true (non-zero) value enables color (allows color images). In medium resolution graphics mode (*mode*=1), a true (non-zero) value will disable color, and a false (zero) value will enable color. Since black and white are the only colors in high resolution graphics (*mode*=2), this parameter will not have much effect in high resolution.

apage	(active page) is an integer expression in the range 0 to 7 for width 40, or 0 to 3 for width 80. It selects the page to be written to by output statements to the screen, and is valid in text mode (*mode*=0) only.
vpage	(visual page) selects which page is to be displayed on the screen, in the same way as *apage* above. The visual page may be different than the active page. *vpage* is valid in text mode (*mode*=0) only. If omitted, *vpage* defaults to *apage*.

If all parameters are valid, the new screen mode is stored, the screen is erased, the foreground color is set to white, and the background and border colors are set to black.

If the new screen mode is the same as the previous mode, nothing is changed.

If the mode is text, and only *apage* and *vpage* are specified, the effect is that of changing display pages for viewing. Initially, both active and visual pages default to 0 (zero). By manipulating active and visual pages, you can display one page while building another. Then you can switch visual pages instantaneously.

If you mix text and graphics in the 40 or 80 column graphics mode and are not using a U.S. keyboard, refer to the GRAFTABL command in the DOS 2.0 book for information regarding additional character support with the Color/Graphics monitor.

Note: There is only one cursor shared between all the pages. If you are going to switch active pages back and forth, you should save the cursor position on the current active page (using POS(0) and CSRLIN), before changing to another active page. Then when you return to the original page, you can restore the cursor position using the LOCATE statement.

TIMER
Function

Purpose: Returns a single-precision number representing the number of seconds that have elapsed since midnight or System Reset. For use in BASIC release 2.0 only.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	

Format: *v* = TIMER

Remarks: Fractional seconds are calculated to the nearest degree possible. TIMER is a read only function.

Example:

```
10 'TIMER returns the number of seconds
20 'since midnight or system reset
30 TIME$="23:59:59"
40 FOR I=1 TO 20
50 PRINT "TIME$= ";TIME$,"TIMER=";TIMER
60 NEXT
RUN

TIME$= 23:59:59      TIMER= 86399.06
TIME$= 23:59:59      TIMER= 86399.11
TIME$= 23:59:59      TIMER= 86399.18
                 .
                 .
                 .
TIME$= 24:00:00      TIMER= 0
TIME$= 00:00:00      TIMER= .05
TIME$= 00:00:00      TIMER= .16
TIME$= 00:00:00      TIMER= .21
Ok
```

NOTES

The returned value is the same as:

CHR$(*type*)+MKI$(VARPTR(*variable*))

You can use VARPTR$ to indicate a variable name in the command string for PLAY or DRAW. For example:

Method One   **Alternative Method**

```
PLAY "XA$;"      PLAY "X"+VARPTR$(A$)
PLAY "O=I;"      PLAY "O="+VARPTR$(I)
```

This technique is mainly for use in programs which will later be compiled.

VIEW Statement

Purpose:	Allows you to define subsets of the viewing surface, called viewports, onto which window contents are mapped. For use in BASIC release 2.0 only.

Versions:	Cassette	Disk	Advanced	Compiler
			***	

Graphics mode only.

Format: VIEW [[*SCREEN*] [(*x1,y1*)−(*x2,y2*) [,[*color*] [,[*boundary*]]]]]

Remarks:

(x1,y1)−(x2,y2)
are the upper-left (*x1,y1*) and the lower-right (*x2,y2*) coordinates of the viewport defined. The *x* and *y* coordinates must be within the actual limits of the screen or an **Illegal function call** error occurs. For more information, see "Specifying Coordinates" under "Graphics Modes" in Chapter 3.

color lets you fill the defined viewport with color. If *color* is omitted, the viewport is not filled. *color* can range from 0 to 3. In medium resolution, this color is the color from the current palette as defined by the COLOR statement. 0 is the background color. The default is the foreground color, color number 3. In high resolution, *color* equal to 0 (zero) indicates black, and the default of 1 (one) indicates white.

boundary lets you draw a boundary line around the viewport (if space is available). If *boundary* is omitted, no boundary is drawn. *boundary* can be a color in the range 0 to 3 as described in *color*.

It is important to note that VIEW sorts the *x* and *y* argument pairs, placing the smallest values for *x* and *y* first. For example:

```
VIEW (1ØØ,1ØØ)-(5,5)
```

becomes:

```
VIEW (5,5)-(1ØØ,1ØØ)
```

Another example:

```
VIEW (-4,4)-(4,-4)
```

becomes:

```
VIEW (-4,-4)-(4,4)
```

All possible pairings of *x* and *y* are valid. The only restriction is that *x1* cannot equal *x2* and *y1* cannot equal *y2*. The viewport cannot be larger than the viewing surface.

If the SCREEN argument is omitted, all points plotted are relative to the viewport. That is, *x1* and *y1* are added to the *x* and *y* coordinate before plotting the point on the screen. For example if:

```
10 VIEW (10,10)-(200,100)
```

is executed, then the point plotted by PSET (0,0),3 is at the actual screen location 10,10.

If the SCREEN argument is included, all points plotted are absolute and may be inside or outside of the screen limits. However, only those points that are within the viewport limits are visible. For example if:

```
10 VIEW SCREEN (10,10)-(200,100)
```

is executed, then the point plotted by PSET (0,0),3 does not appear on the screen because 0,0 is outside of the viewport. PSET (10,10),3 is within the viewport and places the point in the upper-left corner.

VIEW with no arguments defines the entire viewing surface as the viewport. This is equivalent to VIEW (0,0)-(319,199) in medium resolution and VIEW (0,0)-(639-199) in high resolution. When you are in medium resolution, you see SCREEN 1; when you are in high resolution, you see SCREEN 2.

You can define multiple viewports, but only one viewport may be active at a time. RUN and SCREEN will disable the viewports.

VIEW allows you to do scaling by changing the size of your viewport. A large viewport will make your objects large and a small viewport will make your objects small. Scaling with VIEW is similar to zooming with WINDOW. (Refer to "WINDOW Statement" in this chapter.)

Note: When using VIEW, the CLS statement will only clear the current viewport. To clear the entire screen, you must use VIEW to disable the viewports and then use CLS to clear the screen. Using CLS with viewports does not HOME the cursor. Ctrl-Home will HOME the cursor and clear the screen.

Examples: This example demonstrates scaling with VIEW.

```
10 KEY OFF:CLS:SCREEN 1,0:COLOR 0,0
20 WINDOW SCREEN(320,0)-(0,200)
30 GOTO 140
40 '
50 '========================
60 'PICTURE
70    C=1
80    CIRCLE (160,100),60,C,,,5/18
90    CIRCLE (160,100),60,C,,,1
100 '
110 RETURN
120 '========================
130 '
140 GOSUB 60:FOR I=1 TO 1000:NEXT I:CLS
150 'Create the picture
160 VIEW (1,1)-(160,90),,2:GOSUB 60
170 'Make it small
180 END
```

The following example defines four viewports:

```
1Ø SCREEN 1:VIEW:CLS:KEY OFF
2Ø VIEW (1,1)-(151,91),,1
3Ø VIEW (165,1)-(315,91),,2
4Ø VIEW (1,1Ø5)-(151,195),,2
5Ø VIEW (165,1Ø5)-(315,195),,1
6Ø LOCATE 2,4:PRINT "Viewport 1"
7Ø LOCATE 2,25:PRINT "Viewport 2"
8Ø LOCATE 15,4:PRINT "Viewport 3"
9Ø LOCATE 15,25:PRINT "Viewport 4"
1ØØ VIEW (1,1)-(151,91):GOSUB 1ØØØ
2ØØ VIEW (165,1)-(315,91):GOSUB 2ØØØ
3ØØ VIEW (1,1Ø5)-(151,195):GOSUB 3ØØØ
4ØØ VIEW (165,1Ø5)-(315,195):GOSUB 4ØØØ
9ØØ END
1ØØØ CIRCLE (65,5Ø),3Ø,2
1Ø1Ø 'Draw a circle in first viewport
1Ø2Ø RETURN
2ØØØ LINE (45,5Ø)-(9Ø,75),1,8
2Ø1Ø 'Draw a line in second viewport
2Ø2Ø RETURN
3ØØØ FOR D=Ø TO 36Ø:DRAW "ta=d;nu2Ø":NEXT
3Ø1Ø 'Draw spokes in third viewport
3Ø2Ø RETURN
4ØØØ PSET(6Ø,5Ø),2:DRAW "e15;f15;13Ø"
4Ø1Ø 'Draw a triangle in fourth viewport
4Ø2Ø RETURN
```

NOTES

WAIT Statement

Purpose:	Suspends program execution while monitoring the status of a machine input port.			
Versions:	Cassette ***	Disk ***	Advanced ***	Compiler ***

Format: WAIT *port, n*[,*n*]

Remarks: *port* is the port number, in the range 0 to 65535.

n, m are integer expressions in the range 0 to 255.

Refer to the IBM Personal Computer *Technical Reference* manual for a description of valid port numbers (I/O addresses).

The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern.

The data read at the port is XORed with the integer expression *m* and then ANDed with *n*. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If *m* is omitted, it is assumed to be zero.

Example:

```
10 WIDTH "LPT1:",75
20 OPEN "LPT1:" FOR OUTPUT AS #1
 .
 .
 .
6020 WIDTH #1,40
```

In the preceding example, line 10 stores a printer width of 75 characters per line. Line 20 opens file #1 to the printer and sets the width to 75 for subsequent PRINT #1,... statements. Line 6020 changes the current printer width to 40 characters per line.

```
SCREEN 1,0    'Set to med-res color graphics
WIDTH 80      'Go to hi-res graphics
WIDTH 40      'Go back to medium res

SCREEN 0,1    'Go to 40x25 text color mode
WIDTH 80      'Go to 80x25 text color mode
```

WINDOW Statement

Purpose: Allows you to redefine the coordinates of the screen. For use in BASIC release 2.0 only.

Versions:	Cassette	Disk	Advanced	Compiler
			***	

Graphics mode only.

Format: WINDOW [[SCREEN] (*x1,y1*)− (x2,y2)]

Remarks: *(x1,y1), (x2,y2)*

are programmer defined coordinates called *world coordinates*. These coordinates are single-precision, floating-point numbers. They define the world coordinate space that will be mapped into the physical coordinate space, as defined by the VIEW statement. (Refer to "VIEW Statement" in this chapter.)

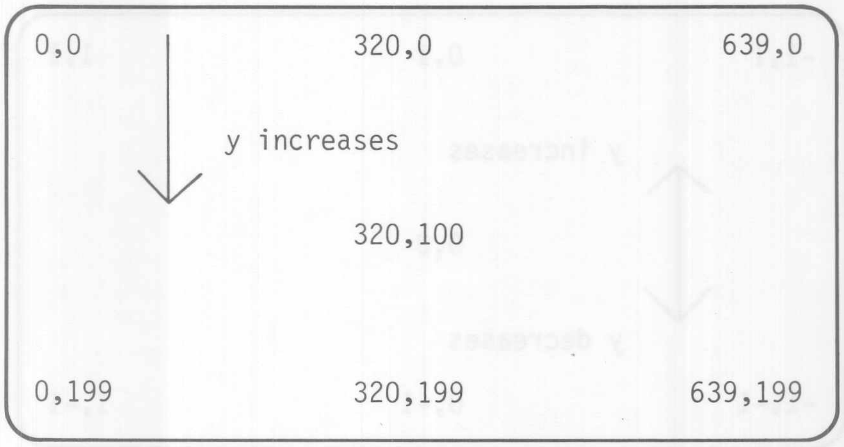
WINDOW allows you to draw objects in space ("world coordinate system") and not be bounded by the logical limits of the screen ("physical coordinate system"). This is done by specifying the world coordinate pairs (x1,y1) and (x2,y2). This rectangular region in the world coordinate space is called a *window*.

BASIC converts the world coordinate pairs into the appropriate physical coordinate pairs for display within the screen.

In the physical coordinate system, if you run the following:

```
NEW
SCREEN 2
```

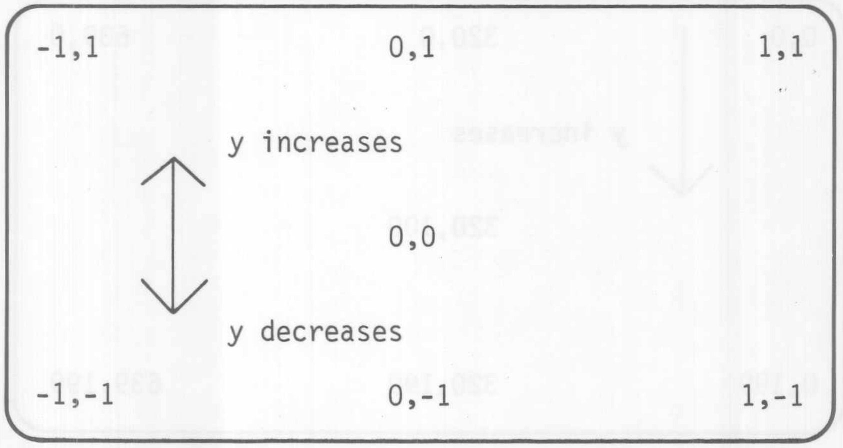
the screen will appear with standard coordinates as:



When the SCREEN attribute is omitted, the screen is viewed in true cartesian coordinates. For example, given:

```
WINDOW (-1,-1)-(1,1)
```

the screen appears as:

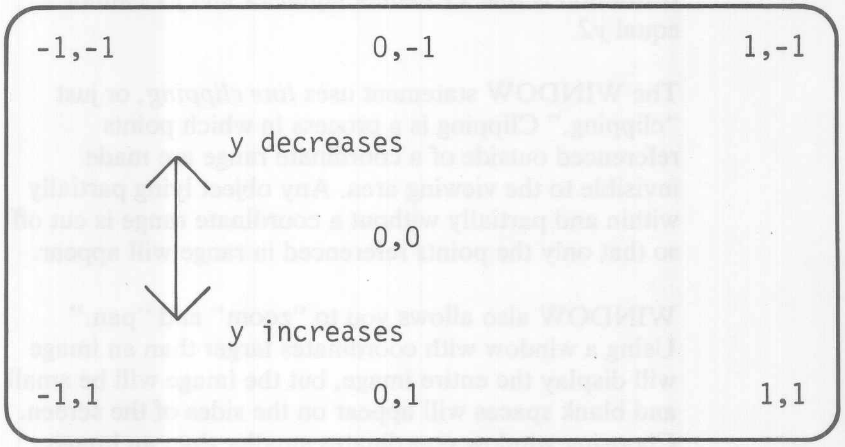


Note that the *y* coordinate is inverted so that *(x1,y1)* is the lower-left coordinate and *(x2,y2)* is the upper-right coordinate.

When the SCREEN attribute is included, the coordinates are not inverted so that *(x1,y1)* is the upper-left coordinate and *(x2,y2)* is the lower-right coordinate. For example:

```
WINDOW SCREEN (-1,-1)-(1,1)
```

defines the screen to look like this:



It is important to note that WINDOW sorts the *x* and *y* argument pairs, placing the smallest values for *x* and *y* first. For example:

```
WINDOW (1ØØ,1ØØ)-(5,5)
```

becomes:

```
WINDOW (5,5)-(1ØØ,1ØØ)
```

Another example:

```
WINDOW (-4,4)-(4,-4)
```

becomes:

```
WINDOW (-4,-4)-(4,4)
```

All possible pairings of *x* and *y* are valid. The only restriction is that *x1* cannot equal *x2* and *y1* cannot equal *y2*.

The WINDOW statement uses *line clipping*, or just "clipping." Clipping is a process in which points referenced outside of a coordinate range are made invisible to the viewing area. Any object lying partially within and partially without a coordinate range is cut off so that only the points referenced in range will appear.

WINDOW also allows you to "zoom" and "pan." Using a window with coordinates larger than an image will display the entire image, but the image will be small and blank spaces will appear on the sides of the screen. Choosing window coordinates smaller than an image forces clipping and allows only a portion of the image to be displayed and magnified. By specifying small and large window sizes, you can zoom in until an object occupies the entire screen, or you can pan out until the image is nothing but a spot on the screen.

RUN, SCREEN, and WINDOW with no attributes will disable any WINDOW coordinates and return the screen to physical coordinates.

Examples: The following example shows clipping using WINDOW.

```
10 SCREEN 2:CLS
20 WINDOW (-6,-6)-(6,6)
30 CIRCLE (4,4),5,1
40 'the circle is large and only part is visible
50 WINDOW (-100,-100)-(100,100)
60 CIRCLE (4,4),5,1 the circle is very small
70 END
```

The following example shows the effect of zooming using WINDOW.

```
10 KEY OFF:CLS:SCREEN 1,0
20
30 GOTO 160
40 '===================
50 'procedure display
60 '
70 LINE (X,0)-(-X,0),,,&HAA00 'create x axis
80 LINE (0,X)-(0,-X),,,&HAA00 'create y axis
90 '
100 CIRCLE (X/2,X/2),R 'circle has radius r
110 FOR P=1 TO 50:NEXT P  'delay loop
120 '
130 RETURN
140 '===================
150 '
160 X=1000:WINDOW (-X,-X)-(X,X):R=20
170 'create a graph with large coord range
180 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
190 '
200 X=60:WINDOW (-X,-X)-(X,X):R=20
210 'smaller coord range increase circle size
220 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
230 '
240 X=100:WINDOW (-5,-5)-(X,X):R=20
250 'modify window to show only portion of axes
260 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
270 '
280 PRINT "...an illustration of zooming..."
290 CLS:T=-50:U=100:X=U
300 FOR P=7 TO 1500:NEXT P
310 FOR I=1 TO 45
320     T=T + 1:U=U - 1:X=X-1:R=20
330     WINDOW (T,T)-(U,U):CLS:GOSUB 50
340 NEXT I
350 END
```

WRITE Statement

Purpose: Outputs data on the screen.

Versions:	Cassette	Disk	Advanced	Compiler
	***	***	***	***

Format: WRITE [*list of expressions*]

Remarks: *list of expressions*

is a list of numeric and/or string expressions, separated by commas or semicolons.

If the list of expressions is omitted, a blank line is output. If the list of expressions is included, the values of the expressions are output on the screen.

When the values of the expressions are output, each item is separated from the last by a comma. Strings are delimited by quotation marks. After the last item in the list is printed, BASIC adds a carriage return/line feed.

WRITE is similar to PRINT. The difference between WRITE and PRINT is that WRITE inserts commas between the items as they are displayed and delimits strings with quotation marks. Also, positive numbers are not preceded by blanks.

Example: This example shows how WRITE displays numeric and string values.

```
10 A=80: B=90: C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
80,90,"THAT'S ALL"
Ok
```

NOTES

APPENDIXES

Contents

NOTES

APPENDIX A. MESSAGES

If BASIC detects an error that causes a program to stop running, an error message is displayed. It is possible to trap and test errors in a BASIC program using the ON ERROR statement and the ERR and ERL variables. (For complete explanations of ON ERROR, ERR and ERL, see "Chapter 4. BASIC Commands, Statements, Functions, and Variables.")

This appendix has two sections. The first section lists alphabetically all of the BASIC error messages with their associated error numbers and includes an explanation of each message. The second section is designed as a quick reference and all message titles are listed in numeric order.

Number	*Message*
73	Advanced Feature Your program used an Advanced BASIC feature while you were using Disk BASIC. Start Advanced BASIC and rerun your program.

Number	*Message*
54	**Bad file mode** You tried to use PUT or GET with a sequential file or a closed file; or to execute an OPEN with a file mode other than input, output, append, or random. Make sure the OPEN statement was entered and executed properly. GET and PUT require a random file. This error also occurs if you try to merge a file that is not in ASCII format. In this case, make sure you are merging the right file. If necessary, load the program and save it again using the A option.
64	**Bad file name** An invalid form is used for the filename with KILL, NAME, or FILES. Check "Naming Files" in Chapter 3 for information on valid filenames, and correct the filename in error.
52	**Bad file number** A statement uses a file number of a file that is not open, or the file number is out of the range of possible file numbers specified at initialization. Or, the device name in the file specification is too long or invalid, or the filename was too long or invalid. Make sure the file you wanted was opened and that the file number was entered correctly in the statement. Check that you have a valid file specification (refer to "Naming Files" in Chapter 3 for information on file specifications).

Number	*Message*
63	**Bad record number** In a PUT or GET (file) statement, the record number is either greater than the maximum allowed (32767) or equal to zero. In BASIC release 2.0, GET and PUT have been enhanced to allow record numbers in the range 1 to 16,777,215 to accommodate large files with short record numbers. Correct the PUT or GET statement to use a valid record number.
17	**Can't continue** You tried to use CONT to continue a program that: ● Halted because of an error, ● Was changed during a break in execution, or ● Does not exist. Make sure the program is loaded, and use RUN to run it.

Number	*Message*
69	**Communication buffer overflow**

A communication input statement was executed, but the input buffer was already full.

You should use an ON ERROR statement to retry the input when this condition occurs. Subsequent inputs try to clear this fault unless characters continue to be received faster than the program can process them. If this happens there are several possible solutions:

- Increase the size of the communications buffer using the /C: option when you start BASIC.
- Implement a "hand-shaking" protocol with the other computer to tell it to stop sending long enough so you can catch up. (See the example in "Appendix F. Communications.")
- Use a lower baud rate to transmit and receive.

25 **Device Fault**

A hardware error indication was returned by an interface adapter. In Cassette BASIC, this only occurs when a fault status is returned from the printer interface adapter.

This message may also occur when transmitting data to a communications file. In this case, it indicates that one or more of the signals being tested (specified on the OPEN "COM... statement) was not found in the specified period of time.

Number	*Message*
57	**Device I/O Error** An error occurred on a device I/O operation. DOS cannot recover from the error. When receiving communications data, this error can occur from overrun, framing, break, or parity errors. When you are receiving data with 7 or less data bits, the eighth bit is turned on in the byte in error.
24	**Device Timeout** BASIC did not receive information from an input/output device within a predetermined amount of time. In Cassette BASIC, this only occurs while the program is trying to read from the cassette or write to the printer. For communications files, this message indicates that one or more of the signals tested with OPEN "COM... was not found in the specified period of time. Retry the operation.

Number	Message
68	**Device Unavailable** You tried to open a file to a device which doesn't exist. Either you do not have the hardware to support the device (such as printer adapters for a second or third printer), or you have disabled the device. (For example, you may have used /C:0 on the BASIC command to start Disk BASIC. That would disable communications devices.) Make sure the device is installed correctly. If necessary, enter the command: `SYSTEM` This returns you to DOS where you can re-enter the BASIC command.
66	**Direct statement in file** A direct statement was encountered while loading or chaining to an ASCII format file. The LOAD or CHAIN is terminated. The ASCII file should consist only of statements preceded by line numbers. This error may occur because of a line feed character in the input stream. Refer to Appendix D, "Converting Programs to IBM Personal Computer BASIC."
61	**Disk full** All diskette storage space is in use. Files are closed when this error occurs. If there are any files on the diskette that you no longer need, erase them; or, use a new diskette. Then retry the operation or rerun the program.

Number	*Message*
72	**Disk Media Error** The controller attachment card detected a hardware or media fault. Usually, this means that the diskette has gone bad. Copy any existing files to a new diskette and re-format the bad diskette. If formatting fails, the diskette should be discarded.
71	**Disk not Ready** The diskette drive door is open or a diskette is not in the drive. Place the correct diskette in the drive and continue the program.
70	**Disk Write Protect** You tried to write to a diskette that is write-protected. Make sure you are using the right diskette. If so, remove the write protection, then retry the operation. This error may also occur because of a hardware failure.
11	**Division by zero** In an expression, you tried to divide by zero, or you tried to raise zero to a negative power. It is not necessary to fix this condition, because the program continues running. Machine infinity with the sign of the number being divided is the result of the division; or, positive machine infinity is the result of the exponentiation. This error cannot be trapped.

Number	*Message*
10	**Duplicate Definition**

You tried to define the size of the same array twice. This may happen in one of several ways:

- The same array is defined in two DIM statements.
- The program encounters a DIM statement for an array after the default dimension of 10 is established for that array.
- The program sees an OPTION BASE statement after an array has been dimensioned, either by a DIM statement or by default.

Move the OPTION BASE statement to make sure it is executed before you use any arrays; or, fix the program so each array is defined only once.

Number	Message
50	**FIELD overflow** A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file in the OPEN statement. Or, the end of the FIELD buffer is encountered while doing sequential I/O (PRINT #, WRITE #, INPUT #) to a random file. Check the OPEN statement and the FIELD statement to make sure they correspond. If you are doing sequential I/O to a random file, make sure that the length of the data read or written does not exceed the record length of the random file.
58	**File already exists** The filename specified in a NAME command matches a filename already in use on the diskette. Retry the NAME command using a different name.

Number	Message
55	**File already open** You tried to open a file for sequential output or append, and the file is already opened; or, you tried to use KILL on a file that is open. Make sure you only execute one OPEN to a file if you are writing to it sequentially. Close a file before you use KILL.
53	**File not found** A LOAD, KILL, NAME, FILES, or OPEN references a file that does not exist on the diskette in the specified drive. Verify that the correct diskette is in the drive specified, and that the file specification was entered correctly. Then retry the operation.
26	**FOR without NEXT** A FOR was encountered without a matching NEXT. That is, a FOR loop was active when the physical end of the program was reached. Correct the program so it includes a NEXT statement.
12	**Illegal direct** You tried to enter a statement in direct mode which is invalid in direct mode (such as DEF FN). The statement should be entered as part of a program line.

Number	Message
5	**Illegal function call** A parameter that is out of range is passed to a system function. The error may also occur as the result of: ● A negative or unreasonably large subscript ● Trying to raise a negative number to a power that is not an integer ● Calling a USR function before defining the starting address with DEF USR ● A negative record number on GET or PUT (file) ● An improper argument to a function or statement ● Trying to list or edit a protected BASIC program ● Trying to delete line numbers which don't exist Correct the program. Refer to "Chapter 4. BASIC Commands, Statements, Functions, and Variables" for information about the particular statement or function.
	Incorrect DOS version The command you just entered requires a different version of DOS from the one you are running.

Number	*Message*
62	**Input past end** This is an end of file error. An input statement is executed for a null (empty) file, or after all the data in a sequential file was already input. To avoid this error, use the EOF function to detect the end of file. This error also occurs if you try to read from a file that was opened for output or append. If you want to read from a sequential output (or append) file, you must close it and open it again for input.
51	**Internal error** An internal malfunction occurred in BASIC. Recopy your diskette. Check the hardware and retry the operation. If the error reoccurs, report to your computer dealer the conditions under which the message appeared.
23	**Line buffer overflow** You tried to enter a line that has too many characters. Separate multiple statements on the line so they are on more than one line. You might also use string variables instead of constants where possible.

Number	*Message*
22	**Missing operand** An expression contains an operator, such as * or OR, with no operand following it. Make sure you include all the required operands in the expression.
1	**NEXT without FOR** The NEXT statement doesn't have a corresponding FOR statement. It may be that a variable in the NEXT statement does not correspond to any previously executed and unmatched FOR statement variable. Fix the program so the NEXT has a matching FOR.
19	**No RESUME** The program branched to an active error trapping routine as a result of an error condition or an ERROR statement. The routine does not have a RESUME statement. (The physical end of the program was encountered in the error trapping routine.) Be sure to include RESUME in your error trapping routine to continue program execution. You may want to add an ON ERROR GOTO 0 statement to your error trapping routine so BASIC displays the message for any untrapped error.

Number	*Message*
4	**Out of data** A READ statement is trying to read more data than is in the DATA statements. Correct the program so that there are enough constants in the DATA statements for all the READ statements in the program.
7	**Out of memory** A program is too large, has too many FOR loops or GOSUBs, too many variables, expressions that are too complicated, or complex painting. You may want to use CLEAR at the beginning of your program to set aside more stack space or memory area.
27	**Out of Paper** The printer is out of paper, or the printer is not switched on. You should insert paper (if necessary), verify that the printer is properly connected, and make sure that the power is on; then, continue the program.
14	**Out of string space** BASIC allocates string space dynamically until it runs out of memory. This message means that string variables caused BASIC to exceed the amount of free memory remaining after housecleaning.

Number	*Message*
6	**Overflow** The magnitude of a number is too large to be represented in BASIC's number format. Integer overflow will cause execution to stop. Otherwise, machine infinity with the appropriate sign is supplied as the result and execution continues. Integer overflow is the only type of overflow that can be trapped. To correct integer overflow, you need to use smaller numbers, or change to single- or double-precision variables. **Note:** If a number is too small to be represented in BASIC's number format, we have an *underflow* condition. If this occurs, the result is zero and execution continues without an error.
75	**Path/file access error** During an OPEN, RENAME, MKDIR, CHDIR or RMDIR operation, an attempt was made to use a path or filename to an inaccessible file. For example, you tried to open a directory or volume identifier; you tried to open a read only file for writing; or you tried to remove the current directory. The operation is not completed.
76	**Path not found** During an OPEN, MKDIR, CHDIR or RMDIR operation, DOS is unable to find the path the way it is specified. The operation is not completed.

Number	*Message*
74	**Rename across disks** You tried to rename a file, but you specified the incorrect disk. The renaming operation is not performed.
20	**RESUME without error** The program has encountered a RESUME statement without having trapped an error. The error trapping routine should only be entered when an error occurs or an ERROR statement is executed. You probably need to include a STOP or END statement before the error trapping routine to prevent the program from "falling into" the error trapping code.
3	**RETURN without GOSUB** A RETURN statement needs a previous unmatched GOSUB statement. Correct the program. You probably need to put a STOP or END statement before the subroutine so the program doesn't "fall" into the subroutine code.
16	**String formula too complex** A string expression is too long or too complex. The expression should be broken into smaller expressions.

Number	*Message*
15	**String too long** You tried to create a string more than 255 characters long. Try to break the string into smaller strings.
9	**Subscript out of range** You used an array element either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts. Check the usage of the array variable. You may have put a subscript on a variable that is not an array, or you may have coded a built-in function incorrectly.
2	**Syntax error** A line contains an incorrect sequence of characters, such as an unmatched parenthesis, a misspelled command or statement, or incorrect punctuation. Or, the data in a DATA statement doesn't match the type (numeric or string) of the variable in a READ statement. When this error occurs, the BASIC program editor automatically displays the line in error. Correct the line or the program.

Number	*Message*
67	**Too many files** An attempt is made to create a new file (using SAVE or OPEN) when all directory entries on the diskette are full, or when the file specification is invalid. If the file specification is okay, use a new formatted diskette and retry the operation.
13	**Type mismatch** You gave a string value where a numeric value was expected, or you had a numeric value in place of a string value. This error may also be caused by trying to SWAP variables of different types, such as single-and double-precision.
8	**Undefined line number** A line reference in a statement or command refers to a line which doesn't exist in the program. Check the line numbers in your program, and use the correct line number.

Number	*Message*
18	**Undefined user function** You called a function before defining it with the DEF FN statement. Make sure the program executes the DEF FN statement before you use the function.
—	**Unprintable error** An error message is not available for the error condition which exists. This is usually caused by an ERROR statement with an undefined error code. Check your program to make sure you handle all error codes which you create.
30	**WEND without WHILE** A WEND is encountered before a matching WHILE was executed. Correct the program so that there is a WHILE for each WEND.
29	**WHILE without WEND** A WHILE statement does not have a matching WEND. That is, a WHILE was still active when the physical end of the program was reached. Correct the program so that each WHILE has a corresponding WEND.

Number	Message
1	NEXT without FOR
2	Syntax error
3	RETURN without GOSUB
4	Out of data
5	Illegal function call
6	Overflow
7	Out of memory
8	Undefined line number
9	Subscript out of range
10	Duplicate Definition
11	Division by zero
12	Illegal direct
13	Type mismatch
14	Out of string space
15	String too long
16	String formula too complex
17	Can't continue
18	Undefined user function
19	No RESUME
20	RESUME without error
22	Missing operand
23	Line buffer overflow
24	Device Timeout
25	Device Fault
26	FOR without NEXT
27	Out of paper
29	WHILE without WEND
30	WEND without WHILE
50	FIELD overflow
51	Internal error

Number	Message
52	Bad file number
53	File not found
54	Bad file mode
55	File already open
57	Device I/O error
58	File already exists
61	Disk full
62	Input past end
63	Bad record number
64	Bad file name
66	Direct statement in file
67	Too many files
68	Device unavailable
69	Communication buffer overflow
70	Disk Write Protect
71	Disk not ready
72	Disk media error
73	Advanced feature
74	Rename across disks
75	Path/file access error
76	Path not found
—	Unprintable error
—	Incorrect DOS Version

NOTES

Appendix H. Hexadecimal Conversion Tables

Hex	Decimal	Hex	Decimal
1	1	10	16
2	2	20	32
3	3	30	48
4	4	40	64
5	5	50	80
6	6	60	96
7	7	70	112
8	8	80	128
9	9	90	144
A	10	A0	160
B	11	B0	176
C	12	C0	192
D	13	D0	208
E	14	E0	224
F	15	F0	240
100	256	1000	4096
200	512	2000	8192
300	768	3000	12288
400	1024	4000	16384
500	1280	5000	20480
600	1536	6000	24576
700	1792	7000	28672
800	2048	8000	32768
900	2304	9000	36864
A00	2560	A000	40960
B00	2816	B000	45056
C00	3072	C000	49152
D00	3328	D000	53248
E00	3584	E000	57344
F00	3840	F000	61440

Binary to Hexadecimal Conversion Table

Binary bit pattern	Hex value	Decimal value
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Some Techniques with Color

Sixteen Background Colors: In text mode, if you are willing to give up blink, you can get all 16 colors (0-15) for the background color. Do the following:

In 40-column width: `OUT &H3D8,8`

In 80-column width: `OUT &H3D8,9`

Character Color in Graphics Mode: You can display regular text characters while in graphics mode. However, if you are not using a U.S. keyboard, refer to the GRAFTABL command in the DOS 2.0 book for information regarding additional character support for the Color/Graphics monitor adapter and other keyboards.

In medium resolution, the foreground color of the characters is color number 3; the background is color number 0.

You can change the foreground color of the characters from 3 to 2 or 1 by performing a:

```
DEF SEG: POKE &H4E, color
```

where *color* is the desired foreground color (1, 2, or 3—0 is *not* allowed). Later PRINTs will use the specified foreground color.

Tips and Techniques

Often there are several different ways you can code something in BASIC and still get the same function. This section contains some general hints for coding to improve program performance.

GENERAL

- **Combine statements where convenient to take advantage of the 255 character statement length.** For example:

  Do

  ```
  100 FOR I=1 TO 10: READ A(I): NEXT I
  ```

  Instead of

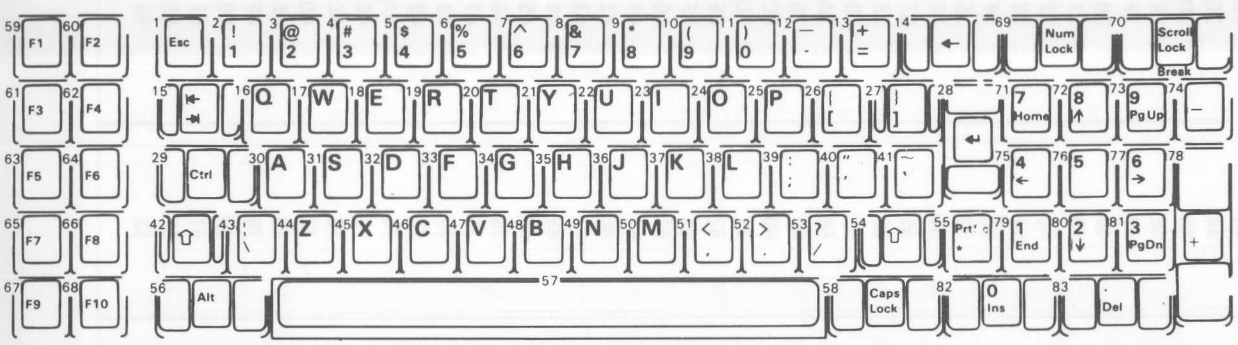
  ```
  100 FOR I=1 TO 10
  110 READ A(I)
  120 NEXT I
  ```

- **Avoid repetitive evaluation of expressions.** If you do the identical calculation in several statements, you can evaluate the expression once and save the result in a variable for use in later statements. For example:

  ```
  Do                  Instead of

  300 X=C*3+D         310 A=C*3+D+Y
  310 A=X+Y           320 B=C*3+D+Z
  320 B=X+Z
  ```

  However, assigning a constant to a variable is faster than assigning the value of another variable to the variable.

Appendix K. Keyboard Diagram and Scan Codes



59 F1 60 F2 1 Esc 2 ! 1 3 @ 2 4 # 3 5 $ 4 6 % 5 7 ^ 6 8 & 7 9 * 8 10 (9 11) 0 12 — - 13 + = 14 ← 69 Num Lock 70 Scroll Lock Break
61 F3 62 F4 15 16 Q 17 W 18 E 19 R 20 T 21 Y 22 U 23 I 24 O 25 P 26 { [27 }] 28 71 7 Home 72 8 73 9 PgUp 74 —
63 F5 64 F6 29 Ctrl 30 A 31 S 32 D 33 F 34 G 35 H 36 J 37 K 38 L 39 : ; 40 " ' 41 ~ ` 75 4 ← 76 5 77 6 → 78
65 F7 66 F8 42 ⇧ 43 \ 44 Z 45 X 46 C 47 V 48 B 49 N 50 M 51 < , 52 > . 53 ? / 54 ⇧ 55 Prt Sc * 79 1 End 80 2 81 3 PgDn +
67 F9 68 F10 56 Alt 57 58 Caps Lock 82 0 Ins 83 . Del

The number to the upper left designates the button position.

Scan Codes

Table 1. **Keyboard Scan Codes**

Key Position	Scan Code in Hex	Key Position	Scan Code in Hex
1	01	43	2B
2	02	44	2C
3	03	45	2D
4	04	46	2E
5	05	47	2F
6	06	48	30
7	07	49	31
8	08	50	32
9	09	51	33
10	0A	52	34
11	0B	53	35
12	0C	54	36
13	0D	55	37
14	0E	56	38
15	0F	57	39
16	10	58	3A
17	11	59	3B
18	12	60	3C
19	13	61	3D
20	14	62	3E
21	15	63	3F
22	16	64	40
23	17	65	41
24	18	66	42
25	19	67	43
26	1A	68	44
27	1B	69	45
28	1C	70	46
29	1D	71	47
30	1E	72	48
31	1F	73	49
32	20	74	4A
33	21	75	4B
34	22	76	4C
35	23	77	4D
36	24	78	4E
37	25	79	4F
38	26	80	50
39	27	81	51
40	28	82	52
41	29	83	53
42	2A		

INDEX